



DIPLOMADO VIRTUAL EN **PROGRAMACIÓN EN JAVA**

Guía didáctica 2: Estructuras de iteración, condición y vectores



Formación Virtual

.....educación sin límites



Competencia específica

Se espera que con los temas abordados en la guía didáctica del módulo 2: Estructuras de iteración, condición y vectores, el estudiante logre la siguiente competencia específica:

- Comprender las diferentes estructuras condicionales y de iteración para almacenamiento de datos en vectores y matrices.



Contenidos temáticos

Los contenidos temáticos para desarrollar en la guía didáctica del módulo 2: Estructuras de iteración, condición y vectores son:

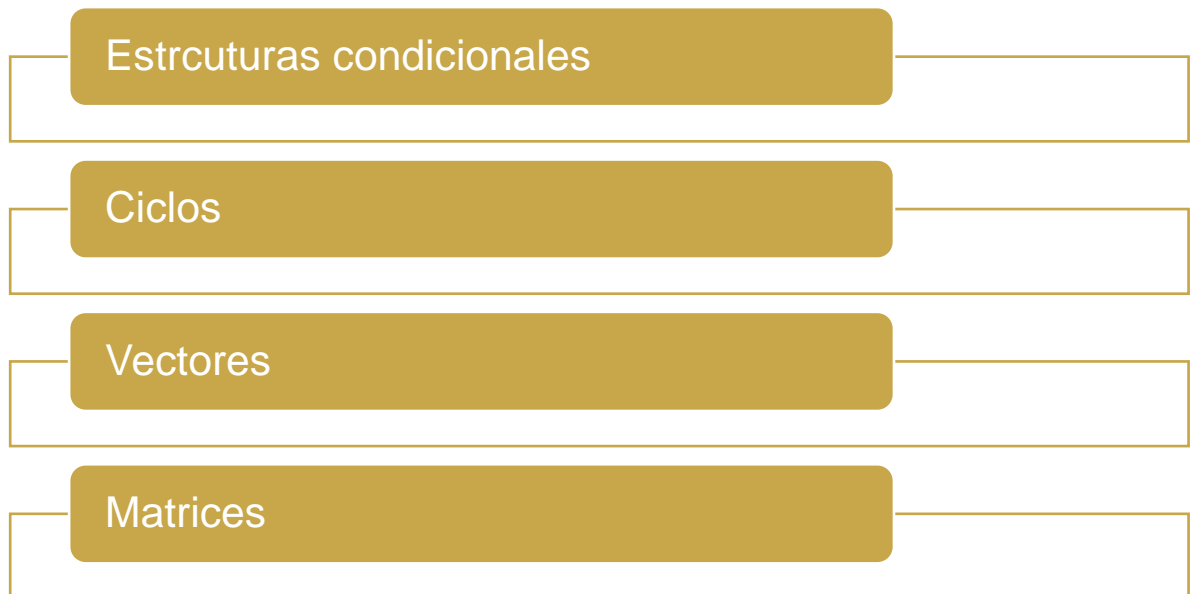


Ilustración 1: caracterización de la guía didáctica.
Fuente: autor.

Tema 1: Estructuras condicionales

¿Qué es una condicional?

Cuando se escribe un programa, se introduce la secuencia de sentencias. Sin sentencias de control del flujo, el intérprete (compilador) ejecuta las sentencias conforme aparecen en el programa de principio a fin. Las sentencias de control de flujo se emplean en los programas para ejecutar sentencias condicionalmente, repetir un conjunto de sentencias o, en general, cambiar el flujo secuencial de ejecución (Garro, 2014).

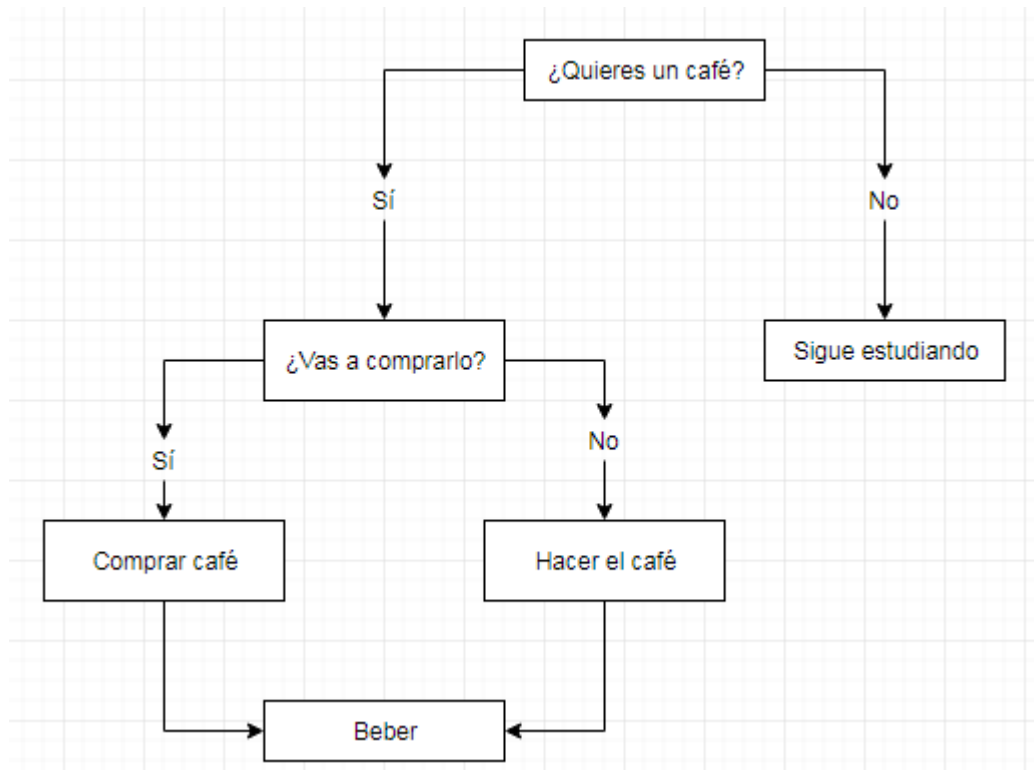


Ilustración 2: ejemplo de condicional: *beber un café*.

Fuente: autor.

Estructura condicional *if*

La estructura condicional más simple en Java es *if*, esta se encarga de evaluar una condición, que para el correcto funcionamiento retornará un valor booleano. En caso de que se cumpla (*true*), se ejecuta el contenido entre las llaves

({}). En caso de que se omita, se ejecuta el código hasta el primer punto y coma (;) por lo tanto, si no se usan los «{}», la condición se aplica solo a la siguiente instrucción al *if*. (Garro, 2014).

```
package Condicionales;

public class Condicionales {

    public static void main(String[] args)
    {
        if(9 < 12)
        {
            System.out.println("9 sí es menor a 12");
        }

        if(4 > 2)
        System.out.println("4 sí es mayor a 2");
    }
}
```

Ilustración 3: condicional *if* por medio de llaves «{}» – punto y coma «;».
Fuente: Eclipse.

Cuando se están diseñando estructuras condicionales, se deben considerar ciertos factores y características:

- Al Java ser tan sensible, es necesario tener en cuenta la forma correcta de diseñar una estructura condicional. **Siempre se debe nombrar el *if* en minúscula, su respectiva condición en paréntesis y un valor opcional (aunque es lo más recomendable) de ejecución dentro de su cumplimiento** (condición verdadera).

```
//if - condición
if (true)
{
    //contenedor de valor en cumplimiento
}
```

- La condición no se limita a una sola sentencia, se pueden tener en cuenta las sentencias necesarias con el requisito de que siempre debe retornar un valor booleano.

```
if(9 < 12 && 2 <= 4)
{
    System.out.println("9 sí es menor a 12");
    System.out.println("2 sí es menor a 4");
}
```

En la condicional se encuentran 2 sentencias para evaluar: (1) $2 < 9$ y (2) $2 \leq 4$ separadas por el operador lógico «&&» (Tabla 7: operadores lógicos – módulo 1) que se encarga de evaluar que ambas sentencias (1) y (2) sean verdaderas para el cumplimiento del «if».

- (1) retornará verdadero debido a que 9 sí es menor que 12.
- (2) retornará verdadero debido a que 2 sí es menor que 4, aunque no es igual (\leq).

Dado que ambas sentencias de condición retornan verdadero, la condición del *if* se cumple. Observemos la siguiente tabla de verdad de operadores lógicos tener dar mayor claridad.

P	Q	P && Q	P Q	!P
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Tabla 1: tabla de verdad.

Fuente: Eclipse.

En la tabla se encuentran dos variables booleanas (1) P y (2) Q, con un valor de inicialización (*true* o *false*) y tres series de casos en los que se pone a prueba la variación a la que se pueden enfrentar. Es necesario tener presente la tabla de operadores lógicos para entender el siguiente ejercicio.

- Evaluación de la primera fila:


```
//Se inicializan ambas variables
boolean p = true;
boolean q = true;
//Primera condición donde P y Q tienen que ser true
//para cumplir el if correctamente.
if(p && q)
{
    //Ejecución en caso de que P y Q sean verdaderas
    System.out.println("P y Q tienen un valor true");
}
//Segunda condición donde P o Q tienen que ser true
//para cumplir el if correctamente.
if(p || q)
{
    //Ejecución en caso de que P o Q sean true
    System.out.println("P o Q tienen un valor true");
}
//Tercera condición donde P al ser true y agregar el operador
// "!" niega el true y lo convierte a false (Sólo en ejecución
//de la condición, la variable P sigue siendo true)
if(!p)
{
    //Ejecución en caso de que P sea true
    System.out.println("P tiene un valor true");
}
```

<terminated> Condicionales [Java Application] C:\

P y Q tienen un valor true

P o Q tienen un valor true

Ilustración 4: condicional *if* con operadores lógicos.

Fuente: Eclipse.

En la fila número 1, el resultado es el esperado; en la serie número 1 con el operador de producto lógico cortocircuito «&&», al cumplirse que (1) y (2) son *true*, la condicional se ejecutará como verdadera. En la serie número 2, con el operador de suma lógica cortocircuito «||», al cumplirse que (1) o (2) son al menos uno *true*, la condicional se ejecutará como verdadera. En la serie número 3, con el operador

de negación «!», al no cumplirse que (1) es *true*, por la negación, la condicional no se ejecutará como verdadera.

- Evaluación de la segunda fila.

```
//Se inicializan ambas variables
boolean p = true;
boolean q = false;
//Primera condición donde P y Q tienen que ser true
//para cumplir el if correctamente.
if(p && q)
{
    //Ejecución en caso de que P y Q sean verdaderas
    System.out.println("P y Q tienen un valor true");
}
//Segunda condición donde P o Q tienen que ser true
//para cumplir el if correctamente.
if(p || q)
{
    //Ejecución en caso de que P o Q sean true
    System.out.println("P o Q tienen un valor true");
}

//Tercera condición donde P al ser true y agregar el operador
// "!" niega el true y lo convierte a false (Sólo en ejecución
//de la condición, la variable P sigue siendo true)
if(!p)
{
    //Ejecución en caso de que P sea true
    System.out.println("P tiene un valor true");
}
```

<terminated> Condicionales [Java Application] C:\

P o Q tienen un valor true

Ilustración 5: condicional *if* con operadores lógicos.

Fuente: Eclipse.

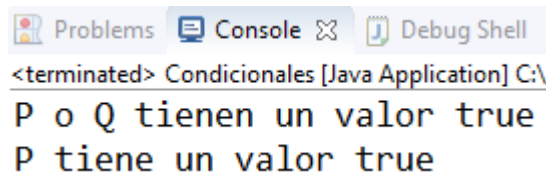
En la fila número 2, el resultado es el esperado; en la serie número 1 con el operador de producto lógico cortocircuito «&&», al no cumplirse que (1) y (2) son *true*, la condicional no se ejecutará como verdadera. En la serie número 2, con el operador de suma lógica cortocircuito «||», al cumplirse que (1) o (2) son al menos uno *true*, la condicional se ejecutará como verdadera. En la serie número 3, con el

operador de negación «!», al no cumplirse que (1) es *true*, por la negación, la condicional no se ejecutará cómo verdadera.

- Evaluación de la tercera fila:

```
//Se inicializan ambas variables
boolean p = false;
boolean q = true;
//Primera condición donde P y Q tienen que ser true
//para cumplir el if correctamente.
if(p && q)
{
    //Ejecución en caso de que P y Q sean verdaderas
    System.out.println("P y Q tienen un valor true");
}
//Segunda condición donde P o Q tienen que ser true
//para cumplir el if correctamente.
if(p || q)
{
    //Ejecución en caso de que P o Q sean true
    System.out.println("P o Q tienen un valor true");
}

//Tercera condición donde P al ser true y agregar el operador
// "!" niega el true y lo convierte a false (Sólo en ejecución
//de la condición, la variable P sigue siendo true)
if(!p)
{
    //Ejecución en caso de que P sea true
    System.out.println("P tiene un valor true");
}
```



Problems Console Debug Shell
<terminated> Condicionales [Java Application] C:\
P o Q tienen un valor true
P tiene un valor true

Ilustración 6: condicional *if* con operadores lógicos.

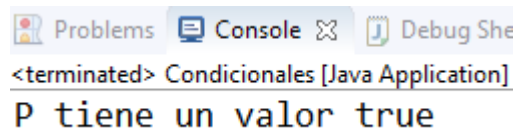
Fuente: Eclipse.

En la fila número 3, el resultado es el esperado; en la serie número 1 con el operador de producto lógico cortocircuito «&&», al no cumplirse que (1) y (2) son *true*, la condicional no se ejecutará como verdadera. En la serie número 2, con el operador de suma lógica cortocircuito «||», al cumplirse que (1) o (2) son al menos

uno *true*, la condicional se ejecutará como verdadera. En la serie número 3, con el operador de negación «!», al cumplirse que (1) es *true*, por la negación, la condicional se ejecutará como verdadera.

- Evaluación de la cuarta fila:

```
//Se inicializan ambas variables
boolean p = false;
boolean q = false;
//Primera condición donde P y Q tienen que ser true
//para cumplir el if correctamente.
if(p && q)
{
    //Ejecución en caso de que P y Q sean verdaderas
    System.out.println("P y Q tienen un valor true");
}
//Segunda condición donde P o Q tienen que ser true
//para cumplir el if correctamente.
if(p || q)
{
    //Ejecución en caso de que P o Q sean true
    System.out.println("P o Q tienen un valor true");
}
//Tercera condición donde P al ser true y agregar el operador
// "!" niega el true y lo convierte a false (Sólo en ejecución
//de la condición, la variable P sigue siendo true)
if(!p)
{
    //Ejecución en caso de que P sea true
    System.out.println("P tiene un valor true");
}
```



Problems Console Debug She
<terminated> Condicionales [Java Application]
P tiene un valor true

Ilustración 7: condicional *if* con operadores lógicos

Fuente: Eclipse.

En la fila número 4, el resultado es el esperado; en la serie número 1 con el operador de producto lógico cortocircuito «&&», al no cumplirse que (1) y (2) son *true*, la condicional no se ejecutará como verdadera. En la serie número 2, con el operador de suma lógica cortocircuito «||», al cumplirse que (1) o (2) son al menos uno *true*, la condicional se ejecutará como verdadera. En la serie número 3, con el

operador de negación «!», al cumplirse que (1) es *true*, por la negación, la condicional se ejecutará cómo verdadera.

Estructura de condicional *e/se*

Hasta ahora, la estructura condicional *if* ha servido para ilustrar los casos en los que una sentencia encerrada en la condicional al momento de retornar un valor booleano *true* permite ejecutar código aparte a raíz del mismo resultado, pero existen casos en que se necesita un complemento al *if*, en los cuales no se cumpla la condicional, ¿qué hará el programa?, ahí entra en juego la estructura condicional *e/se* que permite dentro del programa dictaminar una secuencia de instrucciones en caso de que el *if* no retorne un valor *true* y encerrarlas dentro de {} para su posterior ejecución.

El ejemplo más claro es el de *beber un café* (ilustración 2), en el cual se determinan una serie de casos a raíz de los resultados que la condicional va arrojando.

- Primero está la pregunta/condicional: «¿Quieres un café?».
Existen dos posibles resultados: (1) sí – *true* y (2) no – *false*.
- Si el resultado es (1), es decir *true*, se entra de nuevo en una condicional con la siguiente pregunta/condicional: «¿Vas a comprarlo?».
Existen nuevamente dos resultados: (3) sí – *true* y (4) no – *false*.
- Si el resultado es (3), es decir *true*, se entra a ejecutar «Comprar un café» y, por ende, «beberlo».
- Si el resultado es (4), es decir *false*, se entra a ejecutar «Hacer el café» y, por ende, «Beberlo».
- Si el resultado de «¿Quieres un café?» es (2), es decir *false*, se entra a ejecutar «Sigue estudiando».

Es simple de entender el funcionamiento del *e/se* que juega el papel de la negación, es decir el *false*, y permite fraccionar el código para los casos particulares en que se necesita segmentar el código.

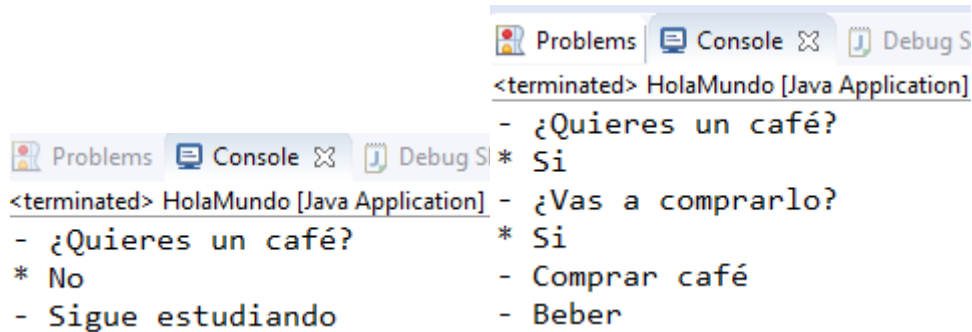
```
//Condicional
if(true)
{
    //Ejecución si es true
    System.out.println("True");
}
else
{
    //Ejecución si es false
    System.out.println("False");
}
```

Ilustración 8: condicional else.

Fuente: Eclipse.

El ejemplo anterior del café, aplicando su sintaxis en código, sería de la siguiente forma:

```
String respuesta1 = "No";
String respuesta2 = "Si";
System.out.println("- ¿Quieres un café?");
System.out.println("* " + respuesta1);
if(respuesta1 == "Si")
{
    System.out.println("- ¿Vas a comprarlo?");
    if(respuesta2 == "Si")
    {
        System.out.println("* " + respuesta2);
        System.out.println("- Comprar café");
    }
    else
    {
        System.out.println("* " + respuesta2);
        System.out.println("- Hacer el café");
    }
    System.out.println("- Beber");
}
else
{
    System.out.println("- Sigue estudiando");
}
```



```

Problems Console Debug S
<terminated> HolaMundo [Java Application]
- ¿Quieres un café?
* Si
- ¿Vas a comprarlo?
* Si
- Comprar café
- Beber

```

Ilustración 9: ejercicio condicional «beber un café».
Fuente: Eclipse.

Hay una particularidad que se debe tener en cuenta a la hora de realizar operaciones booleanas con caracteres, y es la manera de comparar cadenas de texto utilizando el *equals*.

¿Qué es *equals*?

Se encarga de comparar la cadena de texto contra un objeto que devolverá *true* si las cadenas comparadas son iguales; en caso contrario, devolverá *false*.

```

String nombre = "Alejo";

if(nombre.equals("Alejo"))
{
    System.out.println("True");
}

```

Ilustración 10: estructura equals.
Fuente: Eclipse.

Diferencia entre *equals ()* y «==»

En general, los operadores *equals ()* y *==* en Java se usan para comparar objetos, para verificar la igualdad, pero aquí existen algunas diferencias entre los dos. Veamos:

1. La principal diferencia entre el método *.equals ()* y el operador *==* es que uno se aplica a objetos y variables y el otro sobre variables.

2. Se pueden usar los operadores `==` para la comparación de referencias (comparación de direcciones) y el método `.equals ()` para la comparación de contenido. En palabras simples, `==` comprueba si ambos objetos apuntan a la misma ubicación de memoria, mientras que `equals ()` evalúa la comparación de valores en los objetos.

```
String x = new String("Saludo");
String y = new String("Saludo");

System.out.println(x == y);      false
System.out.println(x.equals(y)); true
```

Problems Console [X]
<terminated> Condicionales [Java]

Ilustración 11: relación `equals ()` y `«==»` (1).

Fuente: Eclipse.

```
String x = new String("Saludo");
String y = new String("Saludo");

if(x == y)
{
    System.out.println("true");
}
else
{
    System.out.println("false");
}

if(x.equals(y))
{
    System.out.println("true");
}
else
{
    System.out.println("false");
}
```

Problems Console [X]
<terminated> Condicionales [Java]
false
true

Ilustración 12: relación `equals ()` y `«==»` (2).

Fuente: Eclipse.

Cuando se da utilidad al *e/se*, se debe tener en cuenta que a su aplicación le debe preceder la estructura condicional *if*, y que en caso de cumplirse la segunda (*if*), anula completamente la función del *e/se*.

Siempre que se desee utilizar una sentencia condicional con *e/se*, se debe determinar que la aplicación sí es necesaria. Java admite *e/se {}* sin sentencias que ejecutar, pese a que no es una buena práctica.

Estructura de condicional *else if*

Con base en las dos estructuras condicionales vistas existe una derivación de ambas, cuya utilidad se ve en el momento en el que las condicionales estructuradas como ***if else*** ocupan una validación adicional dentro del ***else***, veámoslo como «sí no se cumple la condición, haga lo siguiente, siempre y cuando se cumpla una nueva condición».

```
if(false)//Primera condición
{

}else if(true)//Segunda condición
{

}else//Tercera condición implícita
{

}
```

La estructura *else if* es de gran importancia cuando se ocupa para revalidar una condición dentro de un *e/se* como la función de un cortafuego, que puede ir de sentencia en sentencia evaluándose hasta encontrar el retorno correcto. Para ejemplificar de mejor forma lo anterior, analicemos el siguiente ejercicio:

En Colombia hay medidas que regulan la velocidad de los automoviles según las zonas donde se encuentre:

- 30 km/h – zonas escolares
- 60 km/h – vías urbanas
- 80 km/h – vías rurales
- 100 km/h – rutas nacionales

Realiza un algoritmo que permita determinar, según una velocidad X, a qué grupo de límites pertenezco y si estoy infringiendo los límites de velocidad.

```
public static void main(String[] args)
{
    int velocidad = 101;

    if(velocidad >= 0 && velocidad <= 30){
        System.out.println("Zonas Escolares");
    }
    else if(velocidad > 30 && velocidad <= 60){
        System.out.println("Vías Urbanas");
    }
    else if(velocidad > 60 && velocidad <= 80){
        System.out.println("Vías Rurales");
    }
    else if(velocidad > 80 && velocidad <= 100){
        System.out.println("Rutas Nacionales");
    }
    else{
        System.out.println("Estás infringiendo los límites de velocidad");
    }
}
```

Ilustración 13: estructura condicional *if else*.

Fuente: Eclipse.

En el ejercicio anterior observamos la aplicación donde el *else if* cumple la función de evaluar otra sentencia previa a la ejecución del *else*; en caso de no estar este *else if*, la condicional se quedaría estancada en únicamente dos posibles resultados (no exactos para el caso):

```
if(velocidad >= 0 && velocidad <= 30){
    System.out.println("Zonas Escolares");
}
else{
    System.out.println("Vías Urbanas");
}
```

No se saldría de la sentencia número 1 o el resultado equívoco de la misma. Además, el uso de *else* repetidamente es erróneo en el lenguaje por la sintaxis, dado que no puede contener dos o más posibles resultados para un solo caso erróneo:

```
if(velocidad >= 0 && velocidad <= 30){  
    System.out.println("Zonas Escolares");  
}  
else{  
    System.out.println("Vías Urbanas");  
}  
else{  
    System.out.println("Vías Rurales");  
}
```

Hay casos en que no se aplica necesariamente el uso del *else if*, y son aquellos en los que solo ocupamos dos resultados un sí o un no únicamente:

```
int numero = 9;  
  
if(numero >= 0){  
    System.out.println("Es positivo");  
}  
else  
{  
    System.out.println("Es negativo");  
}
```

Por definición, se sabe que si el número no es mayor o igual a 0, va a ser negativo, aunque el funcionamiento no se vería afectado en caso de agregarse un *else if*. Veamos:

```
int numero = 9;  
  
if(numero >= 0){  
    System.out.println("Es positivo");  
}  
else if(numero < 0)  
{  
    System.out.println("Es negativo");  
}
```

La segunda sentencia ($\text{numero} < 0$) puede sobrar en caso de que se quiera, en este ejemplo es como un segundo seguro de que se está validando que efectivamente el número sea menor que 0 (negativo), pese a que el *else* lo hace implícitamente.

Estructuras condicionales anidadas

Un paso más allá de las estructuras anteriores están las condicionales anidadas, que permiten ejecutar una sentencia a partir de una condición en forma de ciclo hasta encontrar el retorno esperado (cumplimiento de la condición). Si esta condición se cumple, entonces se ejecuta la sentencia en el programa; en caso de no cumplirse dicha condición, se puede hacer otra condición en el programa para que se cicle, marque error y vuelva a solicitar la información hasta que se cumpla la condición. De no validarse la condición, al igual que en los casos anteriores, se puede definir un caso base donde retornará un valor que se espera (Lasso, 2016).

```
int numero = 9;

if(numero >= 0){
    if(numero > 0 && numero <= 4)
    {
        System.out.println("Bajo valor");
    }
    else if(numero > 4 && numero <= 8)
    {
        System.out.println("Medio valor");
    }
    else if(numero > 8 && numero <= 14)
    {
        System.out.println("Alto valor");
    }
}
else
{
    System.out.println("Es negativo");
}
```

Ilustración 14: estructura condicional anidada.

Fuente: Eclipse.

A diferencia de las estructuras pasadas, las condicionales anidadas proveen un abanico mucho más amplio de posibilidades y combinaciones booleanas para determinar más casos y resultados. En este sentido, hay varios aspectos por recordar cuando se usen condicionales anidadas:

- Utilizar únicamente las condicionales necesarias.

- Evitar redundancias en las sentencias.
- Siempre se debe tratar de tener un retorno en caso de que ninguna condicional se aplique.
- Tener muy presente el uso correcto de las llaves y estar dentro de las condicionales correctas debido al volumen de las mismas.

Switch case (condicional de selección)

La instrucción *switch* es de múltiples vías (condicionales) que proporciona una forma sencilla de enviar la ejecución a diferentes partes del código en función del valor de la expresión. La expresión puede ser tipos de datos primitivos *byte*, *short*, *char* e *int*. Desde JDK7, funciona también con tipos enumerados (*Enum* en Java), la clase *string* y las clases *wrapper*.

```
int num = 2;

switch(num)
{
case 1:
    System.out.print("Número 1");
    break;
case 2:
    System.out.print("Número 2");
    break;
case 3:
    System.out.print("Número 3");
    break;
default:
    System.out.print("Error");
}
```

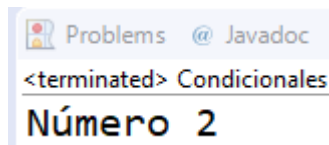


Ilustración 15: estructura *switch case*.

Fuente: Eclipse.

Como en las estructuras pasadas, hay varios factores para tener en cuenta a la hora de desarrollar estructuras condicionales por medio *switch case*. Veamos:

- Los valores duplicados de los *case* no están permitidos.

```
switch(num)
{
  case 1:
    break;
  case 1:
    break;
  default:
    System.out.print("Error");
}
```

- El valor para un *case* debe ser del mismo tipo de datos que la variable en el *switch*.

```
int num = 2; //Variable int
switch(num)
{
  case 1: //Valor int
    break;
  case 2: //Valor int
    break;
  default:
    System.out.print("Error");
}
```

- El valor para un *case* debe ser una constante o un literal. Las variables no están permitidas.

```
switch(num)
{
  case 1:
    break;
  case num:
    break;
  default:
    System.out.print("Error");
}
```

- La declaración *break* se usa dentro del *switch* para finalizar una secuencia de instrucción.

```
int num = 1;

switch(num)
{
case 1:
    //El Switch continúa pese a que
    //la respuesta correcta sea ésta.
case 2:
    break; //El Switch se acaba.
default:
    System.out.print("Error");
}
```

- La declaración *break* es opcional. Si se omite, la ejecución continuará en el siguiente case.

```
switch(num)
{
case 1:
    System.out.println("Número 1");
case 2:
    System.out.println("Número 2");
    break;
default:
    System.out.print("Error");
}
```

- La instrucción *default* es opcional, y debe aparecer al final del *switch* como último caso cumpliendo con similitud la función del *else*.

```
default:
    System.out.print("Error");
}
```

- Se puede realizar cualquier tipo de operación dentro de la declaración del case.

Como se observa, el papel que cumple el *switch case* es muy similar a la función de una condicional anidada con menos instrucciones de código y con una sola sentencia por evaluar.

Break y continue

Las palabras reservadas *break* y *continue* se utilizan en Java para detener completamente un ciclo (*break*) o detener únicamente la iteración actual y saltar a la siguiente (*continue*). Normalmente, si se usa *break* o *continue*, se hará dentro de una

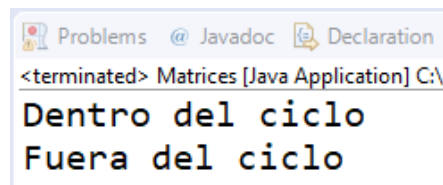
sentencia *if*, que indicará cuándo se debe detener el ciclo al cumplirse o no una determinada condición.

La gran diferencia entre ambos es que *break* detiene la ejecución del ciclo y salta a la primera línea del programa tras el ciclo, y *continue* detiene la iteración actual y pasa a la siguiente iteración del ciclo sin salir de él (a menos que el propio ciclo haya llegado al límite de sus iteraciones).

Nota: la sentencia *continue* debe estar siempre incluida dentro de un ciclo, si no, recibiremos un error de compilación. Mientras que la sentencia *break* se puede encontrar dentro de un bucle o en una sentencia *switch case*.

La sentencia *continue* provoca que únicamente la iteración actual del ciclo más interno detenga su ejecución y que comience inmediatamente la siguiente iteración si el ciclo todavía no ha terminado. Por ejemplo:

```
for(int i = 0; i < 10; i++)
{
    System.out.println("Dentro del ciclo");
    break;
}
System.out.println("Fuera del ciclo");
```

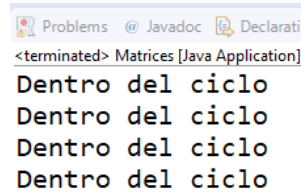


Problems @ Javadoc Declaration
<terminated> Matrices [Java Application] C:\
Dentro del ciclo
Fuera del ciclo

Pese a que la estructura del ciclo *for* esté desarrollada para realizar diez iteraciones, la sentencia *break* dentro de su funcionalidad corta el ciclo en la primera iteración y se sale de él.

En el caso del uso del *continue*, este funciona de la siguiente forma:

```
for(int i = 0; i < 5; i++)
{
    if(i == 3)
    {
        continue;
    }
    System.out.println("Dentro del ciclo");
}
```



```

Problems @ Javadoc Declaration
<terminated> Matrices [Java Application]
Dentro del ciclo
Dentro del ciclo
Dentro del ciclo
Dentro del ciclo

```

El ciclo está declarado para iterar cinco veces en su estructura; la condicional presente en el ciclo determina que, cuando este llegue a la tercera iteración, salte inmediatamente a la siguiente iteración, por lo que en ese caso la instrucción «Dentro del ciclo» no llegará a ser ejecutada y en la salida en consola solo habrá cuatro impresiones.

Omisión de la declaración *break*

Como la declaración *break* es opcional, si se omite el *break*, la ejecución continuará en el siguiente *case*. A veces es deseable tener múltiples *case* sin declaraciones *break* entre ellos. Por ejemplo, determinar en un *switch* case los días laborales de la semana.

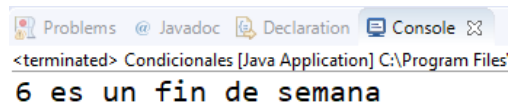
```

int diaNum = 6;
String diaLab;
switch (diaNum)
{
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        diaLab = "dia laborable";
        break;
    case 6:
    case 7:
        diaLab = "fin de semana";
        break;

    default: diaLab= "Tipo de dia invalido";
}

System.out.println(diaNum+" es un "+ diaLab);

```



```

Problems @ Javadoc Declaration Console
<terminated> Condicionales [Java Application] C:\Program Files
6 es un fin de semana

```

Ilustración 16: estructura ejercicio *switch* case.

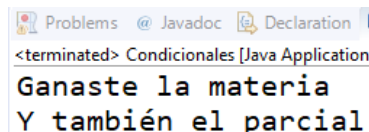
Fuente: Eclipse.

En el ejemplo anterior se observa cómo la omisión del *break* simplifica la creación y ejecución de varios bloques clase resumiendo todos en dos únicos casos de retorno (día laborable – fin de semana).

Switch case anidados

Se puede usar un *switch* como parte de la secuencia de la declaración de un *switch* externo. Esto se llama un *switch* anidado. Como una instrucción de *switch* define su propio bloque, no surgen conflictos entre las constantes de *case* en el *switch* interno y las del *switch* externo. Por ejemplo:

```
int promedio = 4;
int parcial = 4;
switch (promedio)
{
    case 1:
    case 2:
    case 3:
        System.out.println("Perdiste la materia");
        break;
    case 4:
    case 5:
        System.out.println("Ganaste la materia");
        switch(parcial)
        {
            case 1:
            case 2:
            case 3:
                System.out.println("Pero perdiste el parcial");
                break;
            case 4:
            case 5:
                System.out.println("Y también el parcial");
                break;
        }
        break;
    default: System.out.println("Error");
}
```



Problems @ Javadoc Declaration
<terminated> Condicionales [Java Application]
Ganaste la materia
Y también el parcial

Ilustración 17: estructura ejercicio *switch case* anidado.
Fuente: Eclipse.

Switch case con string

Desde JDK7 se puede usar una cadena literal/constante para controlar una declaración *switch*. Usar un modificador basado en cadena/*string* es una mejora con respecto al uso de la secuencia equivalente *if/else*.

```
String nombre = "Ana";
switch (nombre)
{
    case "Diego":
        System.out.println("Hola Diego, lindo día");
        break;
    case "Juan":
        System.out.println("Juan, ¿cómo estás?");
        break;
    case "Stiven":
        System.out.println("Stiven, buena barba");
        break;
    case "Ana":
        System.out.println("Lindo cabello Ana");
        break;
    case "Susana":
        System.out.println("Susana ¿y la familia?");
        break;
    default:
        System.out.println("Error");
}
```

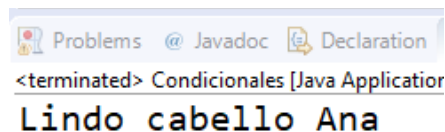


Ilustración 18: estructura ejercicio *switch case string*.

Fuente: Eclipse.

- **Operación costosa:** el *switching* de *string* puede ser más costosa en términos de ejecución que el *switching* de tipos de datos primitivos. Por lo tanto, es mejor activar el *switch* con *string* solo en casos de que los datos de control ya estén en forma de cadena.
- **String no debe ser null:** debemos asegurarnos de que la expresión en cualquier instrucción *switch* no sea nula mientras se trabaja con cadenas,

para evitar que una *NullPointerException* sea lanzada en tiempo de ejecución.

- **Case sensitive – mayúsculas/minúsculas:** la instrucción *switch* compara el objeto *string* en su expresión con las expresiones asociadas con cada etiqueta de case, como si estuviera usando el método *string.equals*; en consecuencia, la comparación de objetos *string* en sentencias *switch* es sensible a mayúsculas y minúsculas.
- **Mejor que if-else:** el compilador Java genera *bytecode* generalmente más eficiente a partir de sentencias *switch* que usan objetos *string* que de sentencias *if-else* anidadas.

Tema 2: Ciclos

Los ciclos son una estructura de control de total importancia para el proceso de creación de un programa. Java y prácticamente todos los lenguajes más populares de la actualidad permiten hacer uso de estas estructuras. Un ciclo en Java permite repetir una o varias instrucciones cuantas veces sea necesario; por ejemplo, si se desean escribir los números del uno al cien, no tendría sentido escribir cien líneas de código mostrando un número en cada una de estas, para eso y para varias cosas más es útil un ciclo. Un ciclo ayuda a ejecutar una tarea repetitiva en una cantidad de líneas muy pequeña y de forma prácticamente automática (Meza, 2018).

Existen diferentes tipos de ciclos o bucles en Java, cada uno con una utilidad para casos específicos. Tenemos las siguientes estructuras:

- Ciclos *for*
- Ciclos *while*
- Ciclos *do while*

Ciclo *for*

Los ciclos *for* (o ciclos para) son una estructura de control cíclica, que permite ejecutar una o varias líneas de código de forma iterativa (o repetitiva), pero teniendo control y conocimiento sobre las iteraciones. En el ciclo *for*, es necesario tener un valor inicial, una condición y valor final, y opcionalmente hacer uso del tamaño del «paso» entre cada iteración del ciclo.

Es decir, un ciclo *for* es una estructura iterativa para ejecutar un mismo segmento de código una cantidad de veces deseada, conociendo previamente un valor de inicio, un tamaño de paso y un valor final para el ciclo.

```
for(int i = 0; i < 10; i++)  
{  
    System.out.println("Número: "+i);  
}
```

Ilustración 19: estructura ciclo *for*.

Fuente: Eclipse.

Estructura del ciclo *for*

La estructura del ciclo *for* se compone de tres elementos fundamentales:

- **Valor de inicio:** determina a partir de qué momento y de qué valor el ciclo inicia las iteraciones.
- **Condicional – valor de parada:** evalúa los casos en que el ciclo realizará las iteraciones y el momento en que terminarán.
- **Valor de incremento - decremento:** realiza un incremento o decremento al valor de inicio para realizar las iteraciones.

```
for(int i = 0; i < 10; i++)  
{  
    //int i = 0; - Valor de inicio  
    //i < 10;    - Condicional y valor de parada  
    //i++;       - Valor de incremento  
}
```

Ilustración 20: estructura ciclo *for* detallada.

Fuente: Eclipse.

Al igual que ocurre con las condicionales, dentro de la declaración de las llaves se ubica respectivamente el código que se desea ejecutar de forma repetitiva, los componentes que definen la estructura se separan como se detallan en la ilustración anterior, por punto y coma «;», esa es la estructura recomendada; aunque hay casos especiales que por declaración se puede diseñar de forma diferente, por ejemplo:

- Creación de valor de inicio fuera del ciclo:

```
int i = 0;          int i = 0;  
for(i = 0; i < 10; i++) for(; i < 10; i++)  
{                      {  
  
                      }  
}
```

- Incremento o decremento dentro del ciclo:

```
for(int i = 0; i < 10; )  
{  
    i++;  
}
```

- Sentencias adicionales dentro de la declaración del ciclo:

```
for(int i = 0; i < 10; System.out.println(i), i++)  
{  
}
```

- Omisión del uso de las llaves:

```
for(int i = 0; i < 10; System.out.println(i), i++);
```

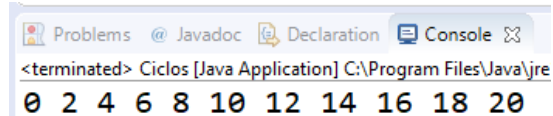
Los casos anteriores son respetados por Java y su uso no tiene ninguna afectación en la estructura de funcionamiento, aunque se recomienda usar la forma tradicional.

Para entender mejor el funcionamiento y aplicación del ciclo, observemos los siguientes ejercicios.

Ejercicios:

- Desarrollar un programa que muestre los números pares entre 0 y 20 de forma ascendente.

```
for(int i = 0; i <= 20; i+=2)  
{  
    System.out.print(i+" ");  
}
```



Problems @ Javadoc Declaration Console
<terminated> Ciclos [Java Application] C:\Program Files\Java\jre
0 2 4 6 8 10 12 14 16 18 20

Para este caso, se conocen de entrada los valores de inicio (0) y cierre/final del ciclo (20); el ejercicio debe mostrar únicamente valores pares, el incremento debe ser diferente al visto en los ejemplos de la estructura (i++) dado que, al declarar el incremento en uno, el ejercicio imprimiría todos los valores que hay entre 0 y 20 (0, 1, 2...18, 19 y 20); el ejercicio funcionaría con un incremento unario en el siguiente caso:

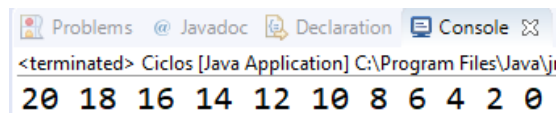
```
for(int i = 0; i <= 20; i++)  
{  
    if(i % 2 == 0)  
    {  
        System.out.print(i+" ");  
    }  
}
```

Dado que se recorre uno a uno todos los valores, pero posteriormente valida por medio de la condicional que únicamente muestre valores divisibles entre 2 (recordemos el uso del módulo «%») y cumple la misma función del primer método, aunque con más líneas de código, lo cual no es eficiente.

Retomando el primer método, el incremento debe ser en (i+=2) para que las iteraciones se realicen adecuadamente e intervalos de dos (2) en dos (2) hasta el valor de parada que será (20).

- ¿Cómo sería el ejemplo anterior descendentemente, es decir, no de 0 a 20, sino de 20 a 0? Así:

```
for(int i = 20; i >= 0; i-=2)
{
    System.out.print(i+" ");
}
```



Problems @ Javadoc Declaration Console
<terminated> Ciclos [Java Application] C:\Program Files\Java\j
20 18 16 14 12 10 8 6 4 2 0

El ejercicio funciona de igual forma con base en los valores que se terminan para el funcionamiento correcto del ejercicio, el valor de inicio cambiaría a (20) para realizar el decremento; la condición de parada se realizará cuando el valor de inicio llegue a (0) y ya no se realiza un incremento gradual de dos en dos, sino un decremento en los mismos valores.

- Desarrollar un programa que permita contar cuántos números entre 1 y 1000 son múltiplos de 7, mostrar el resultado final.

```
int contador = 0;

for(int i = 1; i <= 1000; i++)
{
    if(i % 7 == 0)
    {
        contador++;
    }
}

System.out.println("Los multiplos de 7 entre 1 y 1000 son: "+contador);
```



Problems Javadoc Declaration Console
<terminated> Ciclos [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (10/11/
Los multiples de 7 entre 1 y 1000 son 142

En el desarrollo de este ejercicio entra en uso un concepto muy común en el uso de ciclos: los **contadores**, que no son más que variables que se incrementan consecutivamente según el valor que se determine. La implementación se realiza declarando la variable contadora fuera del ciclo. ¿Por qué?, si se declara dentro del ciclo, pierde completamente su uso y funcionabilidad; el ciclo, al ser una estructura iterativa, declararía la variable las X veces que fueron determinadas en el ciclo, y el valor se reiniciaría por cada iteración, al ser declarado fuera, el uso se limita solo al ciclo.

Los valores del ciclo para este caso son: en el inicio empezará en (1) y el valor de parada será (1000), el incremento será gradual en él para recorrer todos los valores y la condicional interna del ciclo determinará los múltiplos de 7 y realizará el incremento del contador. Al final del ciclo y la condicional por defecto, se encuentra respectivamente el valor de impresión.

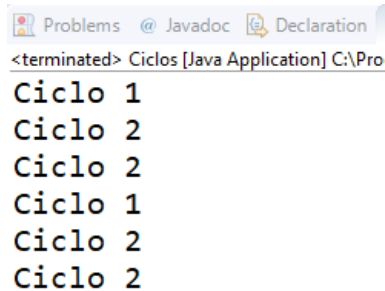
- Desarrollar un problema que imprimir las tablas de multiplicar del 1 al 10 con su respectivo resultado.

¿Se puede tener un ciclo dentro de un ciclo?

```
for(int i = 1; i <= 10; i++)
{
    System.out.println("Tabla de multiplicar "+i);
    for(int j = 1; j <= 10; j++)
    {
        System.out.println(i+" * "+j+" = " + i*j);
    }
}
```

El uso de un ciclo dentro de un ciclo es una operación que se puede realizar con normalidad dentro del programa, y opera de la siguiente forma:

```
for(int i = 0; i < 2; i++)
{
    System.out.println("Ciclo 1");
    for(int j = 0; j < 2; j++)
    {
        System.out.println("Ciclo 2");
    }
}
```



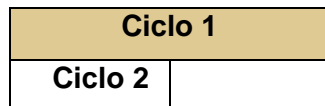
```
<terminated> Ciclos [Java Application] C:\Pro
Ciclo 1
Ciclo 2
Ciclo 2
Ciclo 1
Ciclo 2
Ciclo 2
```

Detenidamente, el uso de un ciclo (1) dentro de otro ciclo (2) condiciona a que este (2) se repita las veces que (1) tenga expresadas. Como se observa en el ejemplo anterior, el ciclo (1) cuenta con dos iteraciones, al igual que el ciclo (2), y se ejecuta de la siguiente forma:

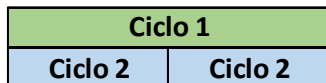
- *Primera ejecución del ciclo (1):*



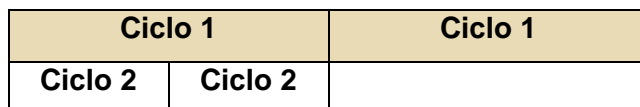
- *Primera ejecución del ciclo (2):*



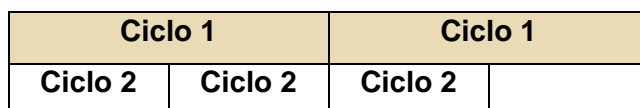
- *Segunda ejecución del ciclo (2):*



- *Segunda ejecución del ciclo (1) – reinicio del ciclo (2):*



- *Primera ejecución del ciclo (2):*



- Segunda ejecución del ciclo (2) – fin del ciclo (1):

Ciclo 1		Ciclo 1	
Ciclo 2	Ciclo 2	Ciclo 2	Ciclo 2

El proceso y el resultado de las tablas de multiplicar son sencillos, en ambos ciclos se tienen los valores y las características para operar. Veamos: ejemplo tabla de multiplicar del 1.

Ciclo 1 - I	Ciclo 2 - J	Resultado
1	1	1
1	2	2
1	3	3
1	4	4
1	5	5
1	6	6
1	7	7
1	8	8
1	9	9
1	10	10

El ciclo (1) con la variable I proporciona la tabla que se está evaluado, y el ciclo (2) con la variable J proporciona los valores que se deben multiplicar, de forma que por cada iteración del ciclo (1) se obtiene el resultado gracias al ciclo (2) con una simple multiplicación.

Ciclo *while*

Los ciclos *while* (o ciclos mientras) son una estructura de control cíclica, que permite ejecutar una o varias líneas de código de forma iterativa (o repetitiva), al igual que el ciclo *for*, pero teniendo control y conocimiento sobre las iteraciones. En el ciclo *for*, era necesario tener un valor inicial, una condición y valor final; en el ciclo *while*, únicamente se tiene control del ciclo por medio de una condicional en la declaración que determina si el ciclo continúa o se detiene, por lo que la sintaxis es más simple.

Es decir, un ciclo *while* es una estructura iterativa para ejecutar un mismo segmento de código, con la particularidad de que en la mayoría de los casos se desconoce la cantidad de veces deseada para iterar por el hecho de no tener un valor de inicio y un valor de parada; puesto que sí se conoce la cantidad de veces que se desea iterar es más simple el uso del ciclo *for*.

```
boolean x = true;

while(x)
{
    System.out.println("Ciclo While");
    x = false;
}

int i = 0;

while(i < 10)
{
    System.out.println(i);
    i++;
}
```

Ilustración 21: estructuras ciclo *while*.

Fuente: Eclipse.

Estructura del ciclo *while*

La estructura del ciclo *while* se compone principalmente de una condición que tiene que tomar un valor booleano (verdadero o falso). Si este valor es verdadero, se ejecutará la sentencia. Concluida esta acción, se vuelve a evaluar la condición. Proseguirán los ciclos hasta que la condición cambie a falso.

Esta es una estructura de iteración preprueba, es decir, primero se evalúa la condición antes de realizar cualquier acción. Si de entrada la condición es falsa, nunca ejecutará el conjunto de sentencias.

Al igual que ocurre con las condicionales, dentro de la declaración de las llaves se ubica respectivamente el código que se desea ejecutar de forma repetitiva; en caso de obtener un resultado verdadero, los componentes que definen la estructura van conforme se detallan en la ilustración anterior: una variable para el control de la condicional, la respectiva y característica condicional (el punto más

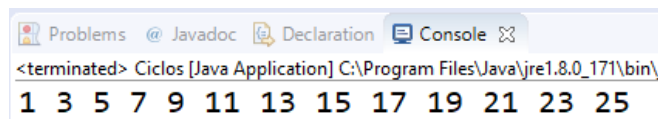
importante) y finalmente la iteración (incremento, decremento o cambio); esta es la estructura recomendada.

Ejercicio:

Desarrollar un programa que imprima los números impares entre 1 y 25.

```
int i = 1;

while(i <= 25)
{
    if(i % 2 != 0)
    {
        System.out.print(i+" ");
    }
    i++;
}
```



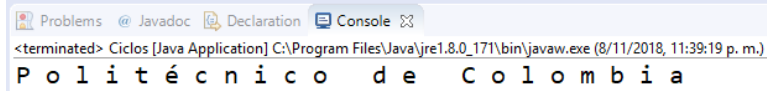
<terminated> Ciclos [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\
1 3 5 7 9 11 13 15 17 19 21 23 25

Este caso tiene un funcionamiento simple y una aplicación similar a los ejercicios desarrollados con la estructura del ciclo *for*; el valor de inicio y final del ciclo lo determina el ejercicio (1 – 25), la condición principal que se debe determinar dentro del *while* debe ser recorrer el ciclo, siempre y cuando se cumpla la condición (ser menor o igual a 25); el incremento, como se especifica en la estructura, debe ser dentro del mismo ciclo y gradualmente en uno (1), es decir (*i++*). Entre las sentencias por ejecutar dentro del ciclo se encuentra la condicional que determina qué número es impar por medio del módulo (%) y el operador de diferencia (! =), dado que un número entero cuyo módulo sea diferente a 0 significa que es un número impar.

- Desarrollar un programa que, dada una palabra, descomponga todos sus caracteres.

```
String palabra = "Politécnico de Colombia";
int i = 0;

while(i < palabra.length())
{
    System.out.print(palabra.charAt(i) + " ");
    i++;
}
```



<terminated> Ciclos [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (8/11/2018, 11:39:19 p. m.)
P o l i t é c n i c o d e C o l o m b i a

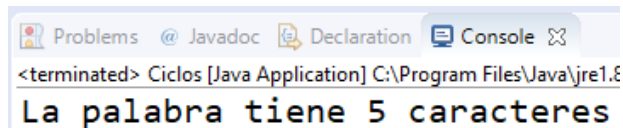
Hay dos métodos nuevos que se aplican para solucionar este ejercicio y que son muy útiles, estos son:

- **Length ()**: devuelve el número de caracteres del *string*, vector y otras estructuras de datos.

Se debe tener en cuenta que el método, al igual que lo hace el *charAt*, reconoce el espacio como un carácter.

```
String palabra = "Diego";

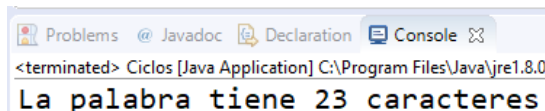
System.out.println("La palabra tiene " + palabra.length() + " caracteres");
```



<terminated> Ciclos [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (8/11/2018, 11:39:19 p. m.)
La palabra tiene 5 caracteres

```
String palabra = "Politécnico de Colombia";

System.out.println("La palabra tiene " + palabra.length() + " caracteres");
```



<terminated> Ciclos [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (8/11/2018, 11:39:19 p. m.)
La palabra tiene 23 caracteres

Palabra																						
P	O	L	I	T	É	C	N	I	C	O		D	E		C	O	L	O	M	B	I	A
23 caracteres																						

- **charAt ()**: devuelve el carácter de una posición del *string*.

Para describir mejor el método *charAt*, este se aplica sobre variables de tipo *string* y se encarga de descomponer la variable en posiciones de 0 a N-1 (N siendo

el número de caracteres), a las cuales se puede acceder por medio de un índice (un número asignado a la posición), por ejemplo:

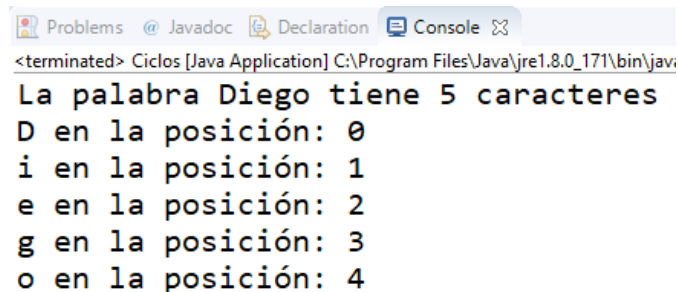
Palabra				
D	I	E	G	O
Posiciones				
0	1	2	3	4

La palabra «Diego» está compuesta por cinco caracteres, por lo que debe tener cinco posiciones asignadas a cada carácter (recordemos que siempre se empieza desde 0, como se observa en la tabla), ¿cómo sería el código?

```
String palabra = "Diego";
int i = 0;

System.out.println("La palabra " + palabra + " tiene "
+ palabra.length() + " caracteres");

while(i < palabra.length())
{
    System.out.println(palabra.charAt(i) + " en la posición: "+i);
    i++;
}
```



```
<terminated> Ciclos [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\jav
La palabra Diego tiene 5 caracteres
D en la posición: 0
i en la posición: 1
e en la posición: 2
g en la posición: 3
o en la posición: 4
```

Lo primero que debemos tener en cuenta es el uso correcto de las posiciones y evitar el desborde en el recorrido de la palabra. El desborde se presenta al buscar una posición que no existe dentro de la palabra. Por ejemplo, como se observa en la tabla, la palabra «Diego» cuenta con cinco caracteres, pero la posición 5 no existe (recordemos el N-1) y arroja lo siguiente al ejecutar:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 5
at java.lang.String.charAt(Unknown Source)
at Clase1.Ciclos.main(Ciclos.java:15)
```

Java se sale del rango que se tiene en la palabra y realiza una excepción de tipo: *StringIndexOutOfBoundsException*.

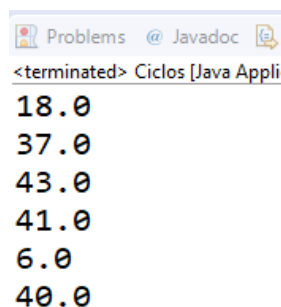
Determinar la condicional dentro del ciclo define el correcto funcionamiento del ejercicio, permite aclarar el caso final de parada del ciclo y evita el desbordamiento en el recorrido de la palabra.

Otra característica para tener presente es el reconocimiento del método *charAt* como un carácter al espacio en blanco.

- Desarrollar un programa que genere números aleatorios entre 1 y 50 y se detenga cuando genere un múltiplo de 10.

Este es uno de los casos en que la implementación del ciclo *while* es más efectiva, dado que no se conoce cuántas veces debe iterar el ciclo, pero sí se conoce el caso de parada. Un ciclo *for* no tendría la misma funcionabilidad que el *while* por su estructura:

```
boolean x = true;
while(x)
{
    double n = Math.ceil(Math.random()*50);
    System.out.println(n);
    if(n % 10 == 0)
    {
        x = false;
    }
}
```



Problems @ Javadoc
<terminated> Ciclos [Java Appli
18.0
37.0
43.0
41.0
6.0
40.0

Para la solución de este ejercicio, se utilizará lo que se conoce como **banderas**, que son variables booleanas que cambian cuando se cumplen ciertas características que se determinen en el código. En este caso, la bandera será la variable X inicializada en verdadero (para realizar la primera iteración del ciclo), y el

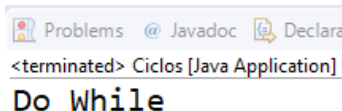
cambio de la bandera a X se dará cuando el número generado sea un múltiplo de 10 (recordemos el uso del módulo). Para generar un número aleatorio, se utiliza como se ha visto, la clase *math* con su método *random* para generar los números entre 1 y 50.

La ejecución del ejercicio termina cuando inmediatamente encuentre el primer múltiplo de 10, dado que se desconoce en qué momento se generará, se utiliza el ciclo *while* y la bandera.

Ciclo *do while*

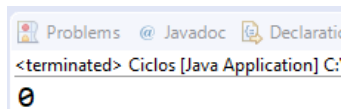
Esta estructura de iteración cumple el mismo objetivo de la estructura *while* con la variante de que el ciclo *do while* ejecuta cuando menos una vez antes de evaluarse la condición del ciclo, por lo que siempre se tendrá una iteración, así el ciclo nunca haya entrado en ejecución.

```
do
{
    System.out.println("Do While");
}
while(false);
```



Problems @ Javadoc Declarati
<terminated> Ciclos [Java Application]
Do While

```
int i = 0;
do
{
    System.out.println(i);
    i++;
}
while(i > 5);
```



Problems @ Javadoc Declarati
<terminated> Ciclos [Java Application] C:
0

Ilustración 22: estructuras ciclo *do while*.

Fuente: Eclipse.

Estructura del ciclo *do while*

La estructura del *do* ciclo *while* cuenta con los mismos componentes del ciclo *while* en cuanto a su forma de funcionar, con la implementación del *do*, que se

encarga de almacenar las sentencias que se ejecutarán en todas las iteraciones para su posterior ejecución.

Esta es una estructura de iteración-prueba, es decir, primero se ejecutan las sentencias del ciclo una vez y después se evalúa la condición antes de realizar las iteraciones esperadas. Si de entrada la condición es falsa, se tendrá una ejecución previa al menos.

Al igual que ocurre con las condicionales, dentro de la declaración de las llaves respectivas del *do*, se ubica el código de las sentencias que se desean ejecutar de forma repetitiva. En caso de obtener un resultado verdadero, los componentes que definen la estructura van como se detalla en las ilustraciones vistas: una variable para el control de la condicional, un bloque encargado de ejecutar las sentencias y, finalmente, la respectiva y característica condicional del ciclo.

Tema 3: Vectores

Los vectores son una estructura de datos que permite almacenar un grupo de datos de un mismo tipo. También son conocidos popularmente como *arrays*. Incluso es habitual llamar matrices a los vectores que trabajan con dos dimensiones.

```
//Forma 1:
int vectorNumeros1[] = new int[10];
//Forma 2:
int []vectorNumeros2 = new int[10];
//Forma 3:
int[] vectorNumeros3 = new int[10];
//Forma 4:
int vectorNumeros4[];
//Forma 5:
int vectorNumeros5[] = {};
//Forma 6:
int vectorNumeros6[] = {9,8,7,6,5,4,3,2,1,0};
//Forma 7:
int vectorNumeros7[] = new int[]{9,8,7,6,5,4,3,2,1,0};
```

Ilustración 23: formas de declarar un vector.

Fuente: Eclipse.

Estructura de un vector

- Los elementos de un vector o *array* se empiezan a numerar en el 0 y permiten gestionar desde una sola variable múltiples datos del mismo tipo.

```
String estudiantes[] = new String[3];

estudiantes[0] = "Juan";
estudiantes[1] = "Stiven";
estudiantes[2] = "Andres";
```

- Al igual que en la aplicación del método *charAt*, en los arreglos, a cada dato almacenado le corresponde un índice, dado que, a partir de los índices, se operan los vectores.

Estudiantes[]		
"Juan"	"Stiven"	"Andres"
Posiciones		
0	1	2

- El tamaño del vector no puede ser redefinido.

```
String estudiantes[] = new String[3];
```

```
estudiantes[0] = "Juan";
estudiantes[1] = "Stiven";
estudiantes[2] = "Andres";
```

```
estudiantes[] = new String[5];
```

- Los vectores pueden ser de N posiciones, siendo N cualquier número entero.

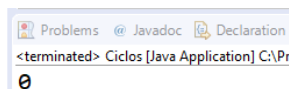
```
String estudiantes[] = new String[300];
double notas[] = new double[30];
```

- Los vectores inicializados sin datos se representan de la siguiente forma por defecto:

- Vectores de números:** se inicializan en 0 los valores de las posiciones.

```
int alturas[] = new int[3];
```

```
System.out.println(alturas[0]);
```

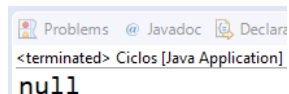


```
Problems @ Javadoc Declaration
<terminated> Ciclos [Java Application] C:\Pr
0
```

- Vectores de cadenas de texto:** se inicializan en vacío o *null* los valores de las posiciones.

```
String nombres[] = new String[3];
```

```
System.out.println(nombres[0]);
```

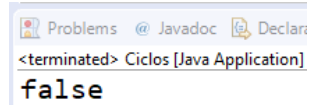


```
Problems @ Javadoc Declaration
<terminated> Ciclos [Java Application]
null
```

- Vectores de booleanos:** se inicializan en *false* los valores de las posiciones.

```
boolean estados[] = new boolean[3];

System.out.println(estados[0]);
```



```
<terminated> Ciclos [Java Application]
false
```

- Los vectores se pueden declarar únicamente o declarar e inicializar, como ocurre con las variables.

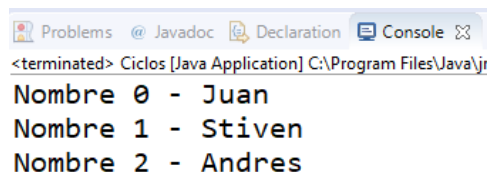
```
boolean estados[] = new boolean[3];
String grupos[] = {"Primero 1", "Once 2"};
```

- Para el uso de vectores, es habitual utilizar ciclos para realizar las operaciones dentro de los mismos, dado que los ciclos permiten definir índices que pueden ser aplicados dentro de los vectores.

```
String nombres[] = new String[3];

nombres[0] = "Juan";
nombres[1] = "Stiven";
nombres[2] = "Andres";

for(int i = 0; i < nombres.length; i++)
{
    System.out.println("Nombre " + i + " - " + nombres[i]);
}
```



```
<terminated> Ciclos [Java Application] C:\Program Files\Java\jre
Nombre 0 - Juan
Nombre 1 - Stiven
Nombre 2 - Andres
```

Ejercicio:

- Desarrollar un programa que, mediante el uso de dos vectores, permita almacenar las cinco notas de un estudiante para calcular su promedio final. Los valores para calcular el promedio se encuentran en un vector aparte que contiene los porcentajes asignados a cada nota.

Porcentaje				
Nota 1	Nota 2	Nota 3	Nota 4	Nota 5

20%	10%	30%	20%	20%
-----	-----	-----	-----	-----

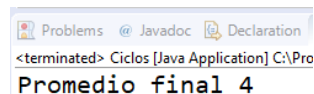
```
double notas[] = new double[5];
int porcentajes[] = {20,10,30,20,20};

notas[0] = 4.5;
notas[1] = 3.2;
notas[2] = 5.0;
notas[3] = 1.5;
notas[4] = 4.3;

double promedio = 0;

for(int i = 0; i < notas.length; i++)
{
    promedio = ( notas[i] * (porcentajes[i]) / 100) + promedio;
}

System.out.println("Promedio final " + Math.round(promedio));
```



Problems @ Javadoc Declaration
<terminated> Ciclos [Java Application] C:\Pro
Promedio final 4

El ejercicio opera de la siguiente forma: se deben implementar dos estructuras de vectores para almacenar los siguientes datos:

- Las notas del estudiante, que en este caso son cinco en un vector *double*, dado que las notas tienen coma flotante.

Notas				
4,5	3,2	5	1,5	4,3
Posiciones				
0	1	2	3	4

- El porcentaje asignado a cada nota, son cinco y asignados a un vector *int*, por el tipo de datos que va a almacenar.

Porcentaje				
Nota 1	Nota 2	Nota 3	Nota 4	Nota 5
20%	10%	30%	20%	20%
Posiciones				
0	1	2	3	4

La fórmula para obtener el promedio es simple, se conoce la cantidad de notas, el porcentaje asignado a cada nota y el valor de las notas; con una simple operación aritmética se puede realizar:

```
//Promedio = (Nota * (PorcentajeNota / 100) + Promedio)
```

Se vuelve a hacer aplicación de los contadores/acumuladores, dentro de la variable promedio se realizarán todas operaciones iteración tras iteración y todos los valores estarán almacenándose dentro de la variable promedio, igual que ocurría en los casos de los contadores con la expresión (i++). Las notas están dentro del vector y por medio del ciclo *for* se accede a los datos de los vectores para poder operar. El comportamiento de la variable promedio sería el siguiente:

Variable promedio				
Iteraciones				
0	1	2	3	4
0,9	1,22	2,72	3,02	3,88
Total				
4				

En la tabla se evalúa la nota multiplicada por el porcentaje dividido por 100 (para trabajar en términos de porcentaje 0,20, 0,30, etc.), y el total redondeado al entero mayor sería 4.

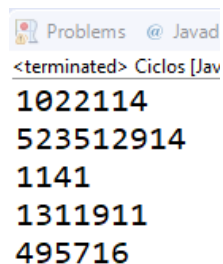
- Desarrollar un ejercicio que, dado un vector de cinco nombres, cambie todas las letras de los nombres por números, de manera que el nombre termine estando compuesto por solo números, por ejemplo:
 - “DIEGO” – “129851”
 - “JUAN” – “6731”

Observemos la siguiente tabla para definir el valor en número de cada letra.

A	B	C	D	E	F	G	H	I	J
1	2	3	4	5	6	7	8	9	10
K	L	M	N	Ñ	O	P	Q	R	S
11	12	13	14	15	16	17	18	19	20
T	U	V	W	X	Y	Z	Caracteres - Valor		
21	22	23	24	25	26	27			

```
String nombres[] = {"JUAN", "EVELIN", "ANA", "MARK", "DIEGO"};
char caracteres[] = {'A','B','C','D','E','F','G','H','I','J','K','L',
                    'M','N','Ñ','O','P','Q','R','S','T','U','V','W','X','Y','Z'};
int numeros[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
                16,17,18,19,20,21,22,23,24,25,26,27};

int x = 0;
String nombreAuxiliar = "";
while(x < 5)
{
    for(int j = 0; j < nombres[x].length(); j++)
    {
        for(int k = 0; k < caracteres.length; k++)
        {
            if(nombres[x].charAt(j) == caracteres[k])
            {
                nombreAuxiliar = nombreAuxiliar+numeros[k];
            }
        }
    }
    nombres[x] = nombreAuxiliar;
    System.out.println(nombres[x]);
    nombreAuxiliar = "";
    x++;
}
```



Problems @ Javac
<terminated> Ciclos [Jav
1022114
523512914
1141
1311911
495716

Este es un ejercicio de un grado de dificultad más alta que los ejercicios pasados, dado que se implementan todos los conocimientos adquiridos hasta el momento para su correcto funcionamiento; hay condicionales, ciclos, vectores, contadores, concatenación y variables.

Partiendo de la tabla proporcionada donde se encuentran los valores numéricos asignados a cada carácter, se determinan 2 vectores con esa información:

- El vector que define los caracteres del abecedario.

```
char caracteres[] = {'A','B','C','D','E','F','G','H','I','J','K','L',  
                    'M','N','Ñ','O','P','Q','R','S','T','U','V','W','X','Y','Z'};
```

- El vector que define los valores numéricos asignados a los caracteres del abecedario.

```
int numeros[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,  
                16,17,18,19,20,21,22,23,24,25,26,27};
```

Es fácil manejar la relación de ambos vectores, dado que tienen la misma longitud, por ende, al valor de la posición 0 del vector de caracteres le corresponde el número 0 del vector de números.

Y, adicionalmente, el vector con los nombres correspondientes por operar, que para el caso práctico son cinco. Además, hay variables del caso que se deben declarar para auxiliar la operación correcta del ejercicio.

- La variable X para administrar el ciclo *while*.

```
int x = 0;
```

- La variable auxiliar nombreAuxiliar que permitirá almacenar el nombre en valores numéricos.

```
String nombreAuxiliar = "";
```

Para el ejercicio, se encuentran definidas tres estructuras de iteración, dos ciclos *for* y un ciclo *while*.

- El ciclo *while* recorrerá el vector de los nombres.

```
while(x < 5)
```

- El ciclo *for* número uno recorrerá los caracteres de cada nombre.

```
for(int j = 0; j < nombres[x].length(); j++)
```

- El ciclo *for* número dos recorrerá los caracteres del vector de caracteres del abecedario ya definido.

```
for(int k = 0; k < caracteres.length; k++)
```

Una condicional para determina qué carácter del nombre le equivale qué número ya establecido:

```
if(nombres[x].charAt(j) == caracteres[k])
{
    nombreAuxiliar = nombreAuxiliar+numeros[k];
}
```

La condicional opera gracias al funcionamiento de los tres ciclos, de la siguiente forma:

- `nombres[x]` representa el valor de un nombre determinado por la iteración del ciclo *while*.
- `nombres[x].charAt(j)` representa los caracteres del nombre que se encuentra en iteración (recordemos: *charAt* permite recorrer todos los caracteres de un *string*, en este caso, por medio de un índice).
- `caracteres[k]` representa los caracteres definidos en el vector del abecedario.

La operación de concatenación que representa el funcionamiento correcto del ejercicio es la siguiente:

```
nombreAuxiliar = nombreAuxiliar+numeros[k];
```

La variable auxiliar declarada e inicializada previamente permite realizar la concatenación de los números de la siguiente forma:

NombreAuxiliar				
Valores				
0	012	01298	0129808	0129808945

La variable va concatenando (no sumando) todos los valores que conforman un nombre hasta llegar al final del recorrido.

```
nombres[x] = nombreAuxiliar;  
System.out.println(nombres[x]);  
nombreAuxiliar = "";  
x++;
```

Finalmente, se reemplaza el nombre de la iteración actual con el nombre creado a partir de números:

```
nombres[x] = nombreAuxiliar;
```

Se imprime el resultado del nombre nuevo:

```
System.out.println(nombres[x]);
```

Se reinicia la variable auxiliar para la próxima iteración:

```
nombreAuxiliar = "";
```

Se realiza el incremento para las próximas iteraciones:

```
x++;
```


Tema 4: Matrices

Las matrices son una estructura de datos que permite almacenar un grupo de datos de un mismo tipo, al igual que ocurre con los vectores, pero de una manera bidimensional, de forma que se representan como una tabla con un vector para filas y otro para columnas.

```
//Forma 1:
int matrizNumeros1[][] = new int[4][5];
//Forma 2:
int [][]matrizNumeros2 = new int[4][5];
//Forma 3:
int[][] matrizNumeros3 = new int[4][5];
//Forma 4:
int matrizNumeros4[][];
//Forma 5:
int matrizNumeros5[][] = {};
//Forma 6:
int matrizNumeros6[][] = {{1,2},{3,9}};
//Forma 7:
int matrizNumeros7[][] = new int[][]{{6,2},{2,7}};
```

Ilustración 24: formas de declarar una matriz.

Fuente: Eclipse.

Estructura de una matriz

- Los elementos de una matriz o *array* bidimensional se empiezan a numerar en el 0,0 (haciendo referencia a las filas y las columnas) y permiten gestionar desde una sola variable múltiples datos del mismo tipo.

```
int numeros[][] = new int[2][2];

numeros[0][0] = 1;
numeros[0][1] = 2;

numeros[1][0] = 6;
numeros[1][1] = 7;
```

- La cantidad de datos que se pueden almacenar los determinan las filas y las columnas por medio de la multiplicación de sus valores, por ejemplo:
 - ✓ Matriz [2][8]: permitirá almacenar 16 valores.
 - ✓ Matriz [3][15]: permitirá almacenar 45 valores.
- A cada dato almacenado dentro de la matriz le corresponde dos índices, uno que determina en qué fila se encuentra el dato y otro en qué columna, dado que, a partir de los índices, se accede a los datos.

Matriz	0	1	2	3
0	Diego	Juan		
1				Frank
2		Stiven		
3			Ana	

En la representación anterior de una matriz 4*4, tiene almacenados cinco datos en las siguientes posiciones:

Diego	[0][0]
Juan	[0][1]
Frank	[1][3]
Stiven	[2][1]
Ana	[3][2]

- El tamaño de la matriz no puede ser redefinido.

```
int numeros[][] = new int[2][2];
```

```
numeros[0][0] = 1;
numeros[0][1] = 2;
```

```
numeros[1][0] = 6;
numeros[1][1] = 7;
```

```
numeros[][] = new int[2][4];
```

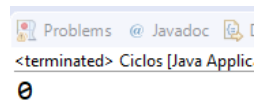
- Las matrices pueden ser de N*M posiciones, siendo N y M cualquier número entero.

```
int numeros[][] = new int[27][92];
String nombres[][] = new String[40][200];
```

- Las matrices inicializadas sin datos se representan de la siguiente forma por defecto:
- Matrices de números:** se inicializan en 0 los valores de las posiciones.

```
int numeros[][] = new int[27][92];

System.out.println(numeros[20][9]);
```

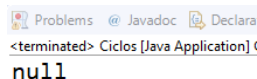


```
Problems @ Javadoc
<terminated> Ciclos [Java Applic
0
```

- Matrices de cadenas de texto:** se inicializan en vacío o *null* los valores de las posiciones.

```
String nombres[][] = new String[27][92];

System.out.println(nombres[20][9]);
```

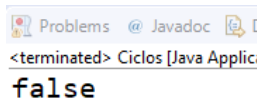


```
Problems @ Javadoc Declara
<terminated> Ciclos [Java Application] (
null
```

- Matrices de booleanos:** se inicializan en *false* los valores de las posiciones.

```
boolean estados[][] = new boolean[27][92];

System.out.println(estados[20][9]);
```



```
Problems @ Javadoc Declara
<terminated> Ciclos [Java Applici
false
```

- Las matrices se pueden declarar únicamente o declarar e inicializar como ocurre con las variables.

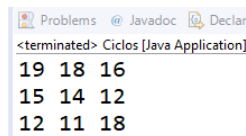
```
int edades[][] = new int[2][3];

int edades[][] = {{19,18,16},{15,14,12},{12,11,8},{22,30,44}};
```

- Para el uso de matrices, es habitual utilizar ciclos para realizar las operaciones dentro de los mismos, dado que los ciclos permiten definir índices que pueden ser aplicados dentro de las matrices, solo que, a diferencia de los vectores, se deben usar dos ciclos para recorrer la matriz, uno se encarga de recorrer las filas y otro de recorrer las columnas. Ejemplo de matriz 3*3:

```
int edades[][] = {{19,18,16},{15,14,12},{12,11,18}};

for(int i = 0; i < 3; i++)
{
    for(int j = 0; j < 3; j++)
    {
        System.out.print(edades[i][j] + " ");
    }
    System.out.println("");
}
```



Problems @ Javadoc Decla...
<terminated> Ciclos [Java Application]
19 18 16
15 14 12
12 11 18

Ejercicios:

- Desarrollar un programa que, dada una matriz de 5*4, almacene números aleatorios entre 0 y 100 en todas sus posiciones. Luego mostrar la matriz en forma de tabla.

```
int numeros[][] = new int[5][4];

for(int i = 0; i < 5; i++)
{
    for(int j = 0; j < 4; j++)
    {
        numeros[i][j] = (int) Math.ceil(Math.random()*100);
    }
}

for(int i = 0; i < 5; i++)
{
    for(int j = 0; j < 4; j++)
    {
        System.out.print(numeros[i][j] + " ");
    }
    System.out.println();
}
```

```
Problems @ Javadoc  
<terminated> Matrices [Java A]  
39 44 94 17  
68 73 19 89  
44 89 25 26  
20 64 10 58  
32 7 54 7
```

Este ejercicio es interesante para determinar correctamente cómo operan las matrices, puede contener menos líneas y ser más eficiente, pero, para el caso de explicación, se desarrolló así:

El ejercicio determina el tamaño de la matriz, que será de 5 – filas * 4 – columnas respectivamente, por lo que podrá almacenar un máximo de veinte valores en su estructura.

Los ciclos son los encargados de operar las matrices, como se describía en su estructura; en la mayoría de los casos, se usan dos para el recorrido: uno que se encarga de las filas y otro de las columnas:

```
for(int i = 0; i < 5; i++)  
{  
    for(int j = 0; j < 4; j++)  
    {  
        numeros[i][j] = (int) Math.ceil(Math.random()*999);  
    }  
}
```

El primer ciclo se encarga de recorrer las filas, cinco para el caso en mención (se utiliza el ciclo *for*, dado que se conoce el número de iteraciones que se deben realizar), y dentro de este un segundo ciclo encargado de recorrer las columnas; cuatro para el caso en mención, como se demostraba el funcionamiento de un ciclo dentro de un ciclo en la estructura *for*, según ciclo opera cuantas veces el primero esté definido. Por ende, el segundo ciclo de cuatro iteraciones se repetirá cinco veces como está definido el primero.

Ciclo 1 - Iteración 1			
Ciclo 2 Iteración 1	Ciclo 2 Iteración 2	Ciclo 2 Iteración 3	Ciclo 2 Iteración 4
Ciclo 1 - Iteración 2			
Ciclo 2 Iteración 1	Ciclo 2 Iteración 2	Ciclo 2 Iteración 3	Ciclo 2 Iteración 4
Ciclo 1 - Iteración 3			
Ciclo 2 Iteración 1	Ciclo 2 Iteración 2	Ciclo 2 Iteración 3	Ciclo 2 Iteración 4
Ciclo 1 - Iteración 4			
Ciclo 2 Iteración 1	Ciclo 2 Iteración 2	Ciclo 2 Iteración 3	Ciclo 2 Iteración 4
Ciclo 1 - Iteración 5			
Ciclo 2 Iteración 1	Ciclo 2 Iteración 2	Ciclo 2 Iteración 3	Ciclo 2 Iteración 4

Tabla 2: ciclos e iteraciones.

Fuente: autor.

Como se muestra en la tabla anterior, por cada iteración del ciclo 1, se obtendrán cuatro iteraciones del ciclo 2, de forma que el resultado de iteraciones totales del ciclo 2 será de 20, el mismo valor de datos esperados para almacenar.

En los parámetros que recibe la matriz para el almacenaje de datos están los dos grupos de corchetes, uno encargado para delimitar las filas con los índices del primer ciclo y el segundo que delimita las columnas a partir del segundo ciclo con sus respectivos índices; para la inserción, dado que son números aleatorios entre un rango especificado, se implementa la clase *math*, con el método *random* para ser generados los números.

Posterior a la inserción de los datos, se encuentra su impresión:

```
for(int i = 0; i < 5; i++)
{
    for(int j = 0; j < 4; j++)
    {
        System.out.print(numeros[i][j] + " ");
    }
    System.out.println();
}
```

El proceso es similar a la inserción: dos ciclos recorren la matriz por filas y columnas e imprimen los respectivos datos en las posiciones determinadas. Para la impresión en forma de tabla, se adiciona una instrucción de impresión vacía para hacer saltos de líneas cada que se recorra una fila.

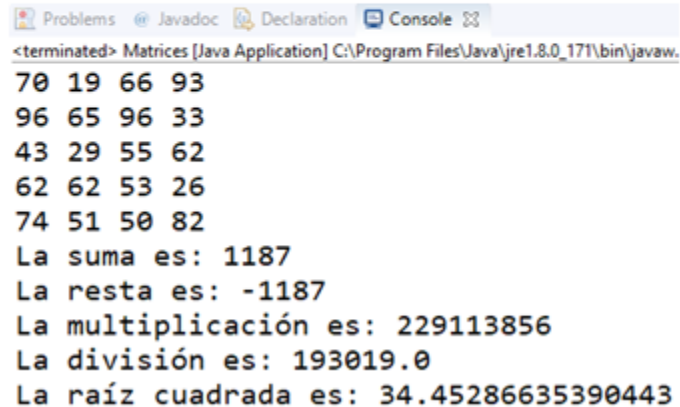
```
for(int j = 0; j < 4; j++)  
{  
    System.out.print(numeros[i][j] + " ");  
}  
System.out.println();
```

- Con base en el ejercicio anterior, realizar las siguientes operaciones:

1. Suma de todos los valores.
2. Resta de todos los valores.
3. Multiplicación de todos los valores.
4. Raíz cuadrada de la suma de todos los valores.
5. División de la multiplicación por la suma.

```
int numeros[][] = new int[5][4];  
int suma = 0, resta = 0;  
int multiplicacion = 1;  
double division;  
double raiz;  
for(int i = 0; i < 5; i++)  
{  
    for(int j = 0; j < 4; j++)  
    {  
        numeros[i][j] = (int) Math.ceil(Math.random()*100);  
    }  
}  
  
for(int i = 0; i < 5; i++)  
{  
    for(int j = 0; j < 4; j++)  
    {  
        suma = suma + numeros[i][j];  
        resta = resta - numeros[i][j];  
        multiplicacion = multiplicacion * numeros[i][j];  
        System.out.print(numeros[i][j] + " ");  
    }  
    System.out.println();  
}
```

```
division = multiplicacion/suma;  
raiz = Math.sqrt(suma);  
  
System.out.println("La suma es: "+suma);  
System.out.println("La resta es: "+resta);  
System.out.println("La multiplicación es: "+multiplicacion);  
System.out.println("La división es: "+division);  
System.out.println("La raíz cuadrada es: "+raiz);
```



```
<terminated> Matrices [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.  
70 19 66 93  
96 65 96 33  
43 29 55 62  
62 62 53 26  
74 51 50 82  
La suma es: 1187  
La resta es: -1187  
La multiplicación es: 229113856  
La división es: 193019.0  
La raíz cuadrada es: 34.45286635390443
```

Partiendo del código del ejercicio anterior, simplemente se deben hacer unas variantes:

Primero se deben declarar las variables en las que se van a almacenar las operaciones que se realizarán basándose en la matriz:

- Una variable entera para la suma inicializada en 0 (para poder realizar la acumulación en base a la matriz).
- Una variable entera para la resta inicializada en 0 (para poder realizar la acumulación en base a la matriz).
- Una variable entera inicializada en 1 para la multiplicación (se inicia en 1 dado que, si se hace en 0, los resultados de las multiplicaciones serían 0).
- Una variable de coma flotante para la división.
- Una variable de coma flotante para la raíz.

La acumulación de datos se puede hacer en dos lugares:

- En la inserción.
- En la impresión.

En la inserción, exactamente después de hacer el guardado de los datos en la matriz, se realizaría la acumulación en las variables declaradas e inicializadas previamente partiendo de los índices del ciclo.

En la impresión, en el mismo recorrido de los ciclos posimpresión o preimpresión, funcionaría de igual forma los acumuladores, así:

```
for(int i = 0; i < 5; i++)
{
    for(int j = 0; j < 4; j++)
    {
        suma = suma + numeros[i][j];
        resta = resta - numeros[i][j];
        multiplicacion = multiplicacion * numeros[i][j];
        System.out.print(numeros[i][j] + " ");
    }
    System.out.println();
}
```

Basándose en los acumuladores, se obtienen los valores finales de la suma, la resta y la multiplicación de todos los valores, ya que con estos se obtendrán:

- La división.
- La raíz.

Dos operaciones aritméticas sencillas para posteriormente mostrar los resultados en pantalla.

```
division = multiplicacion/suma;
raiz = Math.sqrt(suma);

System.out.println("La suma es: "+suma);
System.out.println("La resta es: "+resta);
System.out.println("La multiplicación es: "+multiplicacion);
System.out.println("La división es: "+division);
System.out.println("La raíz cuadrada es: "+raiz);
```



Ejercicio práctico

Ahora que hemos visto un poco acerca de cómo funcionan las condicionales en Java, te sugerimos realizar los siguientes ejercicios que te ayudarán a profundizar en el tema y llenar vacíos: módulo 2 – ejercicios de condicionales simples.

Ya que vimos los tipos principales de condicionales en Java, es importante poner ese conocimiento a prueba. Realiza los siguientes ejercicios con condicionales un poco más complejas: módulo 2 – ejercicios de condicionales complejas.

El *switch* termina siendo una excelente variante al uso de *if/else*, por lo que es importante entrar en detalle de su correcto uso y aplicación. Te recomendamos desarrollar los siguientes ejercicios para fortalecer tus competencias: módulo 2 – ejercicios – *switch case*.

El ciclo *for* ayuda a realizar actividades muy repetitivas y operaciones con volúmenes de datos grandes. Te recomendamos hacer los siguientes ejercicios para fortalecer tus competencias: módulo 2 – ejercicios – ciclo *for*.

El ciclo *while* es otra de las estructuras de iteración que ayuda a la hora de codificar o hacer actividades muy repetitivas. Te sugerimos realizar los siguientes ejercicios para fortalecer tus competencias: módulo 2 – ejercicios – ciclo *while*.

El ciclo *do while* representa una pequeña variante del ciclo *while*. Te recomendamos realizar los ejercicios resueltos con el uso de los ciclos *for* y *while*, pero aplicando la estructura de iteración *do while*.


Las matrices son una excelente alternativa para el almacenamiento de datos y su implementación es sencilla. Te proponemos hacer los siguientes ejercicios: módulo 2 – ejercicios con matrices.

¡Inténtalo!



Referencias bibliográficas

- Eclipse Foundation. (2023). *Eclipse*. Eclipse Foundation. <https://www.eclipse.org/>
- Garro, A. (2014, 15 de abril). Capítulo 5. Sentencias condicionales. En *Java*.
Arkaitz Garro. <https://www.arkaitzgarro.com/java/capitulo-5.html>
- Lasso, D. (2016, 5 de junio). *Cómo funcionan las condicionales anidadas*.
Yosoy.dev. <https://yosoy.dev/como-funcionan-las-condicionales-anidadas/>
- Meza, J. (2018). *Acerca de los bucles o ciclos en Java. Cómo y por qué se usan los bucles o ciclos en Java*. Programar Ya.
<https://www.programarya.com/Cursos/Java/Ciclos>



Esta guía fue elaborada para ser utilizada con fines didácticos como material de consulta de los participantes en el Diplomado Virtual en PROGRAMACIÓN EN JAVA del Politécnico de Colombia, y solo podrá ser reproducida con esos fines. Por lo tanto, se agradece a los usuarios referirla en los escritos donde se utilice la información que aquí se presenta.

GUÍA DIDÁCTICA 2

M2-DV59-GU02

MÓDULO 2: ESTRUCTURAS DE ITERACIÓN, CONDICIÓN Y VECTORES

© DERECHOS RESERVADOS - POLITÉCNICO DE COLOMBIA, 2023
Medellín, Colombia

Proceso: Gestión Académica Virtual
Realización del texto: Diego Palacio, docente
Revisión del texto: Comité de Revisión
Diseño: Comunicaciones

Editado por el Politécnico de Colombia