TEMA 2: USO DE ESTRUCTURAS DE CONTROL

1. INTRODUCCIÓN.

En la unidad anterior has podido aprender cuestiones básicas sobre el lenguaje Java: definición de variables, tipos de datos, asignación de valores, uso de literales, diferentes operadores que se pueden aplicar, conversiones de tipos, inserción de comentarios, etc. Posteriormente, hemos dado los primeros pasos en la solución de algoritmos sencillos, que sólo requieren ejecutar unas instrucciones detrás de otras, sin posibilidad de decidir, según la situación, ejecutar unas u otras sentencias, ni nada parecido. Todo era una ejecución secuencial, una sentencia detrás de otra, sin vuelta atrás, ni saltos de ningún tipo en el orden de ejecución en que estaban escritas. ¿Pero es eso suficiente?

Reflexiona

Piensa en la siguiente pregunta: ¿Cómo un programa puede determinar la aparición en pantalla de un mensaje de ÉXITO o ERROR, según los datos de entrada aportados por un usuario?

Mostrar retroalimentación

No son sólo los datos de entrada aportados por un usuario, existen más variables. Sigue con atención la unidad y lo comprenderás.

Como habrás deducido, con lo que sabemos hasta ahora no es suficiente. Existen múltiples situaciones que nuestros programas deben representar y que requieren tomar ciertas decisiones, ofrecer diferentes alternativas o llevar a cabo determinadas operaciones repetitivamente hasta conseguir sus objetivos.

Si has programado alguna vez o tienes ciertos conocimientos básicos sobre lenguajes de programación, sabes que la gran mayoría de lenguajes poseen estructuras que permiten a los programadores controlar el flujo de la información de sus programas. Esto realmente es una ventaja para la persona que está aprendiendo un nuevo lenguaje, o tiene previsto aprender más de uno, ya que estas estructuras suelen ser comunes a todos (con algunos cambios de léxico o de sintaxis). Si conocías **sentencias de control de flujo** en otros lenguajes, lo que vamos a ver a lo largo de esta unidad te va a sonar bastante, aunque seguro que encuentras alguna diferencia al verlas en Java.

Para alguien que no ha programado nunca, un ejemplo sencillo le va a permitir entender qué es eso de las sentencias de control de flujo.

Piensa en un fontanero (programador), principalmente trabaja con agua (datos) y se encarga de hacer que ésta fluya por donde él quiere (programa) a través de un conjunto de tuberías, codos, latiguillos, llaves de paso, etc. (sentencias de control de flujo). Pues esas **estructuras de control de flujo** son las que estudiaremos, conoceremos su estructura, funcionamiento, cómo utilizarlas y dónde. A través de ellas, al construir nuestros programas podremos hacer que los datos (agua) fluyan por los caminos adecuados para representar la realidad del problema y obtener un resultado adecuado.

Los tipos de **estructuras** de programación que se emplean **para el control del flujo** de los datos, en cualquier lenguaje, son los siguientes:

- ✓ **Secuencial:** compuestas por 0, 1 o más sentencias que se ejecutan en el orden en que han sido escritas. Es la estructura más sencilla y sobre la que se construirán el resto de estructuras. Es de hecho la que hemos usado en los ejemplos y ejercicios de la unidad anterior (un conjunto de sentencias que se ejecutan una detrás de otra).
- ✓ **Selectiva o condicional:** es un tipo de sentencia especial que permite **tomar decisiones**, dependiendo del **valor de una condición** (una expresión lógica). Según la evaluación de la condición se generará un resultado (que suele ser verdadero o falso) y en función de éste, se ejecutará una secuencia de instrucciones u otra (que puede ser no ejecutar ninguna instrucción y pasar a ejecutar la siguiente sentencia detrás del bloque condicional). Las estructuras selectivas (o de selección, o condicionales) podrán ser:
  - **Selectiva simple.**
  - **Selectiva compuesta.**
  - **Selectiva múltiple.**
- ✓ **Iterativa, repetitiva o cíclica:** es un tipo de sentencia especial que permite **repetir la ejecución de una secuencia o bloque de instrucciones** según el resultado de la evaluación de una condición (una expresión lógica). Es decir, la secuencia de instrucciones se ejecutará repetidamente si la condición arroja un valor correcto, en otro caso la secuencia de instrucciones dejará de ejecutarse, y se pasará a ejecutar la siguiente sentencia detrás del ciclo.

Además de las sentencias típicas de control de flujo, en esta unidad haremos una revisión de las **sentencias de salto incondicional**, que, aunque **son altamente desaconsejables en la mayoría de casos, y generalmente resulta innecesario recurrir a ellas**, no está de más conocerlas por si te las encuentras en programas que no han sido escritos por ti.

Posteriormente, analizaremos la mejor manera de llevar a cabo la **depuración** de los programas para localizar errores en nuestro código.

2. USO DE ESTRUCTURAS DE CONTROL.

Este apartado lo utilizaremos para reafirmar cuestiones que son obvias y que de alguna manera u otra las hemos ido viendo explícita o implícitamente a lo largo de la unidad anterior. Vamos a repasarlas como un conjunto de FAQ o "preguntas habitualmente formuladas":

- ✓ **¿Cómo se escribe un programa sencillo?** → Si queremos que un programa sencillo realice instrucciones o sentencias para obtener un determinado resultado, es necesario colocar éstas una detrás de la otra, exactamente en el orden en el que queremos que se ejecuten.
- ✓ **¿Podrían colocarse todas las sentencias una detrás de otra, separadas por puntos y comas en una misma línea?** → En el lenguaje Java (y en muchos otros) puede hacerse, pero no es muy recomendable. Cada sentencia debería estar escrita en una línea diferente. De esta manera tu código será mucho más legible y la localización de errores en tus programas será más sencilla y rápida, lo que redundará en menor tiempo de desarrollo y de mantenimiento, lo que reducirá los costes. De hecho, cuando se utilizan herramientas de programación, los errores suelen asociarse a un número o números de línea.
- ✓ **¿Puede una misma sentencia ocupar varias líneas en el programa?** → Sí. Existen sentencias que, por su tamaño, pueden generar varias líneas. Pero siempre finalizarán con un punto y coma.

- ✓ **¿En Java todas las sentencias se terminan con punto y coma?** → Efectivamente. Si detrás de una sentencia ha de venir otra, pondremos un punto y coma. Escribiendo la siguiente sentencia en una nueva línea. Pero en algunas ocasiones, sobre todo cuando utilizamos estructuras de control de flujo, detrás de la cabecera de una estructura de este tipo no debe colocarse punto y coma. No te preocupes, lo entenderás cuando analicemos cada una de ellas.
- ✓ **¿Qué es la sentencia nula o sentencia vacía en Java?** → La sentencia nula es una línea que no contiene ninguna instrucción y en la que sólo existe un punto y coma. Como su nombre indica, esta sentencia no hace nada. Se puede colocar en cualquier sitio donde la sintaxis del lenguaje exija que vaya una sentencia, pero no queramos que se haga nada. Normalmente su uso puede evitarse, usando una lógica adecuada al construir las estructuras de control de flujo. Pero es importante que la conozcas por si alguna vez te la encuentras en algún programa no hecho por ti (o porque realmente llegaras a necesitarla).
- ✓ **¿Qué es un bloque de sentencias?** → Es un conjunto de sentencias que se encierra entre llaves y que se ejecutaría como si fuera una única sentencia. Sirve para agrupar sentencias y para clarificar el código. Los bloques de sentencias son utilizados en Java en la práctica totalidad de estructuras de control de flujo, clases, métodos, etc. La siguiente tabla muestra dos formas de construir un bloque de sentencias. ¿Cuál de las dos crees que es más clara y que por tanto se recomienda usar?

Bloque de sentencias en una sola línea	Bloque de sentencias de varias líneas

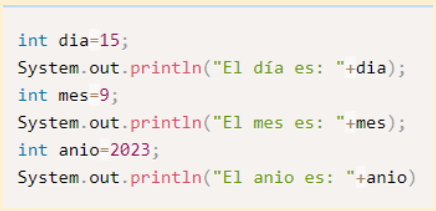
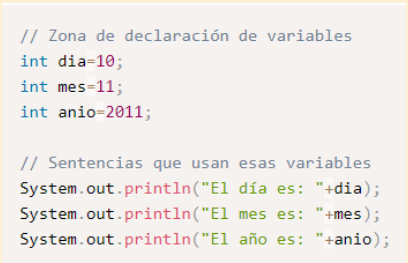
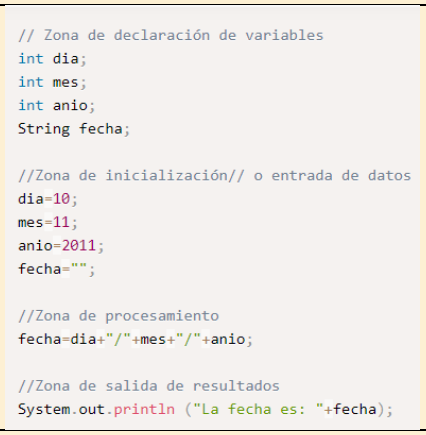
`{ sentencia_1; sentencia_2; ... ; sentencia_N; }`	`{`
	`  sentencia_1;`
	`  sentencia_2;`
	`  . . .`
	`  sentencia_N;`
	`}`

- ✓ **En un bloque de sentencias, ¿éstas deben estar colocadas con un orden exacto?** → En ciertos casos sí, aunque si al final de su ejecución se obtiene el mismo resultado, podrían ocupar diferentes posiciones en nuestro programa.
- ✓ **¿Es importante documentar los programas?** → Claro que sí. Más adelante veremos que se pueden documentar de diversas formas, pero una de ellas es escribir el código con claridad (autodocumentación), usando indentación adecuada de las sentencias, eligiendo nombres descriptivos para variables, métodos y clases, incluyendo comentarios explicativos, etc.
- ✓ **¿Y me tengo que preocupar de las faltas de ortografía y de gramática?** → Lo principal es preocuparte de la propia sintaxis del lenguaje para que todo funcione correctamente, eso es cierto. Pero es MUY importante que las cadenas literales que se incluyan en el código sean correctas, porque son la primera, y a veces la única impresión que el cliente/usuario obtendrá de la aplicación, y si ve faltas de ortografía o frases incoherentes o complicadas de entender, difícilmente lo vas a convencer de que al escribir en un lenguaje de programación eres una persona cuidadosa y fiable a la hora de escribir código. Además, parte de la documentación se genera con herramientas automáticas a partir de los comentarios, así que la corrección en los mismos también es fundamental.

Debemos asegurarnos de que **cualquier texto que aparezca en el código de un programa**, ya sea en un **comentario** o en una **cadena entre comillas**, está correctamente escrito. Los textos deben aparecer **sin erratas ni faltas de ortografía** y **sin errores gramaticales**, así como con un **discurso coherente** y con un **uso adecuado de los signos de puntuación**.

DEBES CONOCER…

Los tres códigos fuente obtienen el mismo resultado, pero la organización de las sentencias que los componen es diferente entre ellos.

Bloque de sentencias para ejecución secuencial.	**Bloque de sentencias con declaración de variables.**	**Bloque de sentencias con el código organizado en declaración de variables, entrada de datos, procesamiento y salida.**
		
En este primer ejemplo, las sentencias están colocadas en orden secuencial.	En este segundo archivo, se declaran al principio las variables necesarias. En Java no es imprescindible hacerlo así, pero sí que antes de utilizar cualquier variable ésta debe estar previamente declarada. Aunque la declaración de dicha variable puede hacerse en cualquier lugar de nuestro programa.	En este tercer archivo, podrás apreciar que se ha organizado el código en las siguientes partes: declaración de variables, petición de datos de entrada, procesamiento de dichos datos y obtención de la salida. Este tipo de organización está más estandarizada y hace que nuestros programas ganen en legibilidad y claridad.

Construyas de una forma o de otra tus programas, en Java debes tener en cuenta siempre las siguientes premisas:

- ✓ Declara cada variable antes de utilizarla.
- ✓ Inicializa con un valor cada variable la primera vez que la utilices.
- ✓ No es recomendable usar variables no inicializadas en nuestros programas, pueden provocar errores o resultados imprevistos.

Podrás observar que la tercera forma de organizar las sentencias de un programa es la que hemos estado recomendando a lo largo de la unidad anterior en cuanto aprendimos a escribir algo de código. Se trata de la "***plantilla estándar de programa principal***" que llevamos ya usando para este módulo desde hace algunas semanas. Esa **es la que vamos a utilizar durante todo el curso siempre que nos sea posible**.

2.1 ÁMBITO O ALCANCE DE UNA VARIABLE.

Hasta el momento habrás podido observar que las variables siempre se definen dentro de un bloque (entre llaves { ... }). Por ahora no hemos tenido mucho problema porque la estructura de todos nuestros programas hasta el momento tenía un único bloque dentro de un mecanismo llamado **main**:

```
public class ProgramaEjemplo {

    public static void main(String[] args) {
        ...

        // Aquí dentro, entre las llaves, estamos escribiendo el código de nuestros programas...

        ...
    }

}
```

Más adelante, cuando aprendamos a implementar **métodos** dentro de una clase, volveremos a ver este tema con más profundidad al estudiar las variables locales, pero dado que a partir de ahora vamos a empezar a utilizar más de un bloque dentro de nuestros programas, es importante que introduzcamos el concepto de **ámbito, contexto o alcance de una variable** (también conocido como *scope*). El ámbito de una variable es la zona de código en la que esa variable "existe", es decir, donde puede ser utilizada tanto para obtener su valor como para cambiarlo (suponiendo que no haya sido definida como constante **final**).

Ese ámbito está definido por el bloque encerrado entre llaves donde ha sido declarada la variable. En el momento en que esa llave se cierre, la variable será eliminada y cualquier sentencia que intente hacer referencia a ella será considerada como un error, pues esa variable ya no existe, de manera que no podréis compilar ni ejecutar ese programa.

Dado que en cuanto comencemos a trabajar con las estructuras de control no secuenciales (las condicionales y las repetitivas) vamos a tener un montón de bloques, algunos de ellos unos dentro de otros y en otros casos en bloques independientes, vamos a ver un ejemplo de bloques donde se vayan creando y destruyendo variables para que observéis cómo funciona. Tened en cuenta que cualquier variable declarada dentro de un bloque (inicio de la llave "{") dejará de existir en cuanto se cierre el bloque (fin de la llave "}") donde fue declarada.

```
public class ProgramaEjemploBloques {

    public static void main(String[] args) {
        // Bloque principal (0)
        int num1;
        num1 = 10;

        System.out.println("En bloque principal (0)");
        System.out.println("num1= " + num1);
        System.out.println();
        {
            // Bloque 0.1
            System.out.println("En bloque 0.1");
            int num2;
            num1++;
            num2 = 20;
            System.out.println("num1= " + num1);
            System.out.println("num2= " + num2);
            System.out.println();
        }
        System.out.println("En bloque principal (0)");
        System.out.println("num1= " + num1);
        //System.out.println("num2= " + num2); // Error  (LÍNEA 23)
        System.out.println();
        {
            // Bloque 0.2
            System.out.println("En bloque 0.2");
```

```
            int num2;
            num1++;
            num2 = 20;
            System.out.println("num1= " + num1);
            System.out.println("num2= " + num2);
            System.out.println();
            {
                // Bloque 0.2.1
                System.out.println("En bloque 0.2.1");
                int num3 = 30;
                num1++;
                num2++;
                System.out.println("num1= " + num1);
                System.out.println("num2= " + num2);
                System.out.println("num3= " + num3);
                System.out.println();
            }
            System.out.println("En bloque 0.2");
            System.out.println("num1= " + num1);
            System.out.println("num2= " + num2);
            //System.out.println("num3= " + num3);// Error  LÍNEA 48
            System.out.println();
        }
        System.out.println("En bloque principal (0)");
        System.out.println("num1= " + num1);
        //System.out.println("num2= " + num2);// Error LÍNEA 53
        //System.out.println("num3= " + num3);// Error LÍNEA 54
        System.out.println();
    }
}
```

Ejercicio Resuelto

Fijándote en el código anterior, dado que se trata de un programa donde todo el flujo es secuencial (cada sentencia se ejecuta una detrás de otra) te debería resultar sencillo intuir cuál va a ser el resultado de su ejecución, ¿verdad?

Intenta escribir en un papel, sin ejecutar el programa, cuál crees que sería el resultado de la ejecución del programa anterior.

La ejecución debería dar como resultado algo similar a lo siguiente:

```
En bloque principal (0)
num1= 10

En bloque 0.1
num1= 11
num2= 20

En bloque principal (0)
num1= 11

En bloque 0.2
num1= 12
num2= 20

En bloque 0.2.1
num1= 13
num2= 21
```

```
num3= 30


En bloque 0.2
num1= 13
num2= 21


En bloque principal (0)
num1= 13
```

Ahora bien, ¿por qué crees que las sentencias de las líneas 23, 48, 53 y 54 han sido marcadas como errores?

¿Qué explicación darías para el caso de la línea 23?

```
System.out.println("num2= " + num2)
```

En el caso de la **línea 23** debemos tener en cuenta que la variable **num2** fue declarada en la línea 19, dentro del bloque "etiquetado" como 0.1 (abierto en la línea 11), de manera que en cuanto ese bloque se cierre (línea 20), cualquier variable que hubiera sido declarada en su interior dejará de tener sentido en el resto del programa, salvo que sea de nuevo declarada. Por tanto, si se intenta acceder a esa variable fuera del bloque, el compilador de Java nos indicaría que se está produciendo un error léxico (una palabra que no conoce porque no es ni una palabra del lenguaje Java ni ninguna variable declarada por nosotros en ese ámbito).

¿Y para el caso de la línea 48?

```
System.out.println("num3= " + num3);
```

En este caso se trata de una variable declarada dentro del bloque que hemos "identificado" como 0.2.1. Sin embargo, ese bloque se cerró en la línea 44, de manera que cualquier intento de acceso a esa variable en cualquier otra parte del programa que no sea en el interior del bloque nos producirá un error de compilación, nuevamente un error de tipo léxico (una palabra o símbolo "desconocido").

¿Y respecto a la línea 53?

```
System.out.println("num2= " + num2);
```

Aquí nos vuelve a suceder lo mismo que con la **línea 23**: **num2** es una variable que no existe en el contexto o ámbito del bloque principal (o bloque 0), de manera que se trata del acceso una variable desconocida y que producirá un error léxico al intentar compilar.

En este caso hemos tenido dos contextos anteriores (bloques 0.1 y 0.2) donde sí ha existido una variable que se llamaba así, pero no en el contexto o ámbito del bloque 0, donde esa variable no está declarada y por tanto no existe. Fíjate que los nombres de variable pueden repetirse en ámbitos diferentes, pues en realidad se trata de variables diferentes (son "creadas" al iniciarse el ámbito o bloque y "destruidas" al cerrarse).

Lo que no puedes tener es variables que cuyo nombre coincida con el nombre de otra variable dentro de un bloque mayor en el que se encuentre incluido. Es decir, dentro del bloque 0.1 no puedes tener una variable que se llame **num1**, pues el bloque 0.1 está dentro del bloque 0 y por tanto ya existe una variable llamada **num1**. Sin embargo, en el bloque 0.2 sí has podido declarar una variable **num2** porque ni en ese bloque ni en cualquier otro bloque más amplio en el que se encuentre incluido existe una variable con ese nombre. Es decir, cuanto más "pequeño" o "específico" sea el contexto (esté dentro de más llaves), más posibilidades hay de que tengamos más variables declaradas, mientras que cuanto más amplio sea (menos llaves), menos declaraciones de variables habrá.

3. ESTRUCTURAS DE SELECCIÓN O CONDICIONALES.

Al principio de la unidad nos hacíamos esta pregunta: **¿Cómo un programa puede determinar la aparición en pantalla de un mensaje de ÉXITO o ERROR, según los datos de entrada aportados por un usuario?**

Ésta y otras preguntas se nos plantean en múltiples ocasiones cuando desarrollamos programas.

¿Cómo conseguimos que nuestros programas puedan tomar decisiones? lo haremos a través de las **estructuras de selección**. Estas estructuras constan de una **condición** que se evalúa para ver si toma el valor verdadero o falso, y de un conjunto de secuencias de instrucciones, que se ejecutarán o no dependiendo de si la condición se evaluó como verdadera o como falsa. Puede haber dos bloques de instrucciones, de forma que si es verdadera la condición se ejecuta el primer bloque y si es falsa, se ejecuta el otro bloque.

Por ejemplo, si el valor de una variable es **mayor o igual que 5** (condición verdadera) se **imprime por pantalla la palabra APROBADO** (primer grupo de sentencias, en este caso una sola) y **si es menor que 5** (condición falsa) se **imprime SUSPENSO** (segundo grupo de sentencias, también con una sola en este caso). Para este ejemplo, la comprobación del valor de la variable será lo que nos permite decidir qué camino tomar y cuál es la siguiente instrucción a ejecutar. La impresión de la palabra APROBADO será una secuencia de instrucciones y la impresión de la palabra SUSPENSO será otra. Cada secuencia estará asociada a cada uno de los resultados que puede arrojar la evaluación de la condición.

Las estructuras de selección se dividen en:

- ✓ Estructura de **selección simple** o estructura **if.**
- ✓ Estructura de **selección compuesta** o estructura **if-else**.
- ✓ Estructura de selección basada en el operador condicional, representado en Java por **?**.
- ✓ Estructura de **selección múltiple** o estructura **switch**.

3.1 ESTRUCTURA CONDICIONAL SIMPRE: IF.

La estructura de control **if** es una **estructura de selección** o **estructura selectiva** o **estructura condicional**. Os la podéis encontrar con todos estos nombres. Permite condicionar la ejecución entre dos sentencias (o dos bloques de sentencias) dependiendo de la evaluación de una expresión lógica (condición). Al comprobar una condición podamos tomar dos caminos alternativos (bloques de sentencias) dependiendo de si esa condición se evaluó como verdadera (**true**) o como falsa (**false**). La representación en diagrama de flujo sería la de la imagen de la derecha, que es bastante explicativa:

- ✓ **Si la expresión que se evalúa es verdadera**, **se ejecuta la secuencia de instrucciones 1**.
- ✓ **Si es falsa, se ejecuta la secuencia de instrucciones 2**. Esta rama, a veces, puede no contener ninguna sentencia a ejecutar. Este caso es el que conocemos como **estructura condicional simple**. Es el que vamos a estudiar en este apartado.

En ambos casos, una vez finalizada la ejecución del bloque de sentencias 1 o 2, el flujo continúa con la siguiente sentencia que haya tras esta estructura condicional.

En lenguaje natural (o pseudocódigo), eso se expresaría como:

Estructura condicional simple.(Si-entonces)

```
Si expresion Entonces
   secuencia_1
Fin Si
```

Funcionamiento:
La secuencia de instrucciones `secuencia_1` se ejecuta si y solo si en el caso de que la `expresion` se evalúe como verdadera. No se hace nada en caso contrario, simplemente "se omite" la ejecución de `secuencia_1`.

Fíjate que la palabra **Entonces** se indica para delimitar con claridad dónde termina la expresión que se va a evaluar y dónde empieza la secuencia de instrucciones del primer bloque.

La estructura **if** puede presentarse en Java de las siguientes formas:

Estructura if simple.

Sintaxis para el caso de ejecución condicional de una sola rama con una sola sentencia

```
if (expresion_logica)
    sentencia_1;
```

Sintaxis para el caso de ejecución condicional de una sola rama con un bloque de sentencias

```
if (expresion_logica) {
    sentencia_1;
    sentencia_2;
    . . .
    sentencia_N;
}
```

Funcionamiento:
Si la evaluación de la `expresion_logica` ofrece un resultado verdadero, se ejecuta la `sentencia_1` o bien el bloque de sentencias asociado. Si el resultado de dicha evaluación es falso, no se ejecutará ninguna instrucción asociada a la estructura condicional.

La mejor de manera de entender el funcionamiento de esta estructura es mediante un ejemplo sencillo. Imagina que dependiendo del contenido de una variable entera llamada **valor** se muestre por pantalla el texto **"El valor es negativo"** o bien no se muestre nada. Bastaría con escribir algo así:

```
1  int valor = -10;
2  System.out.println ("El programa está ejecutándose."); // Esta línea se ejecuta en cualquier caso
3  if (valor < 0)
4      System.out.println("El valor es negativo."); // Esta línea solo se ejecuta si valor < 0
5  System.out.println ("El programa sigue ejecutándose."); // Esta línea se ejecuta en cualquier caso
```

En este caso se mostraría por pantalla lo siguiente:

```
El programa está ejecutándose.
El valor es negativo.
El programa sigue ejecutándose.
```

Dado que el contenido de la variable valor es negativo, se ejecuta la línea 4. Si no, se habría saltado directamente a la 5. Veámoslo con otro ejemplo. Si la variable valor contuviera un número que no fuera negativo, como es en el siguiente caso:

```
1  int valor = 0;
2  System.out.println ("El programa está ejecutándose."); // Esta línea se ejecuta en cualquier caso
3  if (valor < 0)
4      System.out.println("El valor es negativo."); // Esta línea solo se ejecuta si valor<  0
5  System.out.println ("El programa sigue ejecutándose."); // Esta línea se ejecuta en cualquier caso
```

El resultado de la ejecución de este fragmento de código no mostraría por pantalla el texto **"El valor es negativo"**, aunque sí se ejecutarían las sentencias que hay justo antes (línea 2) y justo después (línea 5), pues no forman parte de la estructura condicional:

```
El programa está ejecutándose.
El programa sigue ejecutándose.
```

Como habrás podido observar, al tener una única sentencia dentro del "bloque" de la estructura **if**, no es necesario encerrar ese bloque entre llaves. Si queremos incluir más de una sentencia, sí que tendremos que encerrar todo ese bloque de sentencias entre llaves.

Si en el caso anterior, además de mostrar un texto por pantalla, queremos hacer algo más, tendremos que encerrar todas esas sentencias entre llaves para que el compilador de Java sepa que son todas esas sentencias las que se tienen que ejecutar cuando la condición de la estructura **if** sea evaluada como **true**. Si no se encierra el bloque entre llaves, se entenderá que tan solo debe ejecutarse la primera sentencia y que el resto de sentencias sí se ejecutarán en cualquier caso (tanto si el resultado de la evaluación de la expresión lógica es **true** como si es **false**).

Por ejemplo, si además de mostrar el mensaje por pantalla, queremos poner la variable valor a 0 y mostrar otro mensaje más por pantalla, tendremos que encerrar todas esas sentencias entre llaves formando un bloque. Lo haríamos de la siguiente manera:

```
1  int valor = -10;
2  System.out.println ("El programa está ejecutándose."); // Esta línea se ejecuta en cualquier caso
3  if (valor < 0) {
4      System.out.println ("El valor es negativo."); // Esta línea solo se ejecuta si valor < 0
5      valor= 0; // Esta línea solo se ejecuta si valor < 0
6      System.out.println ("El valor ha sido reseteado a cero."); // Esta línea solo se ejecuta si valor < 0
7  }
8  System.out.println ("El programa sigue ejecutándose."); // Esta línea se ejecuta en cualquier caso
```

En este caso el resultado obtenido en pantalla sería:

```
El programa está ejecutándose.
El valor es negativo.
El valor ha sido reseteado a cero.
El programa sigue ejecutándose.
```

Si el valor no hubiera sido negativo y hubiéramos tenido, por ejemplo:

```
1  int valor = 0;
2  System.out.println ("El programa está ejecutándose."); // Esta línea se ejecuta en cualquier caso
3  if (valor < 0) {
4      . . .
```

El resultado habría sido:

```
El programa está ejecutándose.
El programa sigue ejecutándose.
```

Sin embargo, si no hubiéramos encerrado entre llaves el bloque de las sentencias que deben ejecutarse cuando el resultado de evaluar valor < 0 sea true, el código tendría el siguiente aspecto:

```
1  int valor = 0;
2  System.out.println ("El programa está ejecutándose."); // Esta línea se ejecuta en cualquier caso
3  if (valor < 0)
4      System.out.println ("El valor es negativo."); // Esta línea solo se ejecuta si valor < 0
5      valor= 0; // Esta línea se ejecuta en cualquier caso
6      System.out.println ("El valor ha sido reseteado a cero."); // Esta línea se ejecuta en cualquier caso
7  System.out.println ("El programa sigue ejecutándose."); // Esta línea se ejecuta en cualquier caso
```

Y el resultado de la ejecución habría sido:

```
El programa está ejecutándose.
El valor ha sido reseteado a cero.
El programa sigue ejecutándose.
```

Donde podemos ver cómo las líneas 5 y 6 se han ejecutado independientemente de que el resultado de evaluar la expresión lógica valor < 0 haya sido true o false. Y eso probablemente no es lo que nosotros deseábamos que sucediera. Debes tener cuidado con esto, pues se trata de un error bastante habitual cuando se empieza a programar. Por mucha indentación que añadamos, si no encerramos el bloque entre llaves, Java considerará que el bloque es de una única sentencia y que el resto de sentencias están fuera de la estructura condicional. Es decir, que es como si hubiéramos escrito:

```
1  int valor = 0;
2  System.out.println("El programa está ejecutándose."); // Esta línea se ejecuta en cualquier caso
3  if (valor < 0) {
4      System.out.println("El valor es negativo."); // Esta línea solo se ejecuta si valor < 0
5  }
6  valor = 0; // Esta línea se ejecuta en cualquier caso
7  System.out.println("El valor ha sido reseteado a cero."); // Esta línea se ejecuta en cualquier caso
8  System.out.println("El programa sigue ejecutándose."); // Esta línea se ejecuta en cualquier caso
```

Reflexiona

Para indicar **un bloque de ejecución dentro de una estructura de control condicional**, ¿utilizamos la **indentación** o las **llaves**? ¿O ambas? ¿Que sentido tiene cada una?

Mostrar retroalimentación

Lo habitual es utilizar ambas:

1. las **llaves** se utilizan para que el compilador sepa cuáles son las sentencias que forman el bloque;
2. la **indentación** se usa para que a los seres humanos nos sea más sencillo distinguir que ese bloque está "dentro" de la estructura de control y que por tanto solamente se ejecutará si se dan ciertas condiciones. Pero para que eso realmente sea así el bloque, además de indentarse, tendrá que estar encerrado entre llaves.

Es decir, que el primer caso es para indicar el bloque propiamente dicho al compilador de Java, mientras que el segundo es para **facilitar la legibilidad del código fuente** a las personas. Aunque ambos aspectos son muy importantes y debemos cumplirlos, el primero (el **uso de las llaves**) es imprescindible para que Java sepa qué es lo que queremos hacer.

Ejercicio Resuelto

Es probable que recuerdes que en la unidad pasada se os proporcionó una herramienta llamada **Scanner** que nos servía para recibir entradas desde el teclado, de tal manera que podíamos hacer cosas como:

```
Scanner teclado = new Scanner(System.in); // No olvides incluir al principio una sentencia import java.util.Scanner
int numero;
System.out.print("Introduzca un número entero: ");
numero = teclado.nextInt();
```

Teniendo en cuenta todo esto, ¿cómo podríamos implementar un programa **leyera desde teclado un número entero** y nos indicara si se trata de un **número par**?

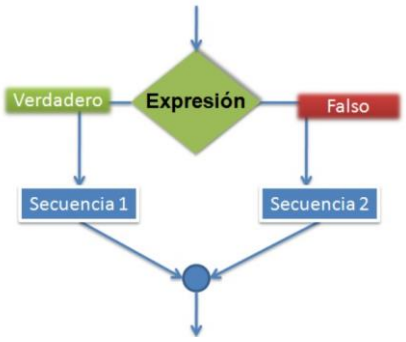
Se trataría de un programa con dos partes:

1. La lectura del número (entrada de datos);
2. La comprobación del número. Si el número resulta ser par (el resultado de aplicarle la operación módulo es cero), entonces se muestra un mensaje por pantalla. En caso contrario no se hace nada y el programa sigue con su ejecución justo después de la sentencia condicional.

```
// Primera parte: entrada de datos
Scanner teclado = new Scanner(System.in);
int numero;
System.out.print("Introduzca un número entero: ");
numero = teclado.nextInt();


// Segunda parte: procesamiento (comprobación) y salida de resultados (mensaje por pantalla si es par)
if (numero % 2 == 0) {  // Si el resto de la división entera entre 2 es cero, el número es par
    System.out.println("El número es par.");
}
```

Una vez leído de teclado el número, lo siguiente que hacemos es calcular el resto de la división entera del número entre 2 (mediante el operador módulo representado por el símbolo **%** en Java) y a continuación comparamos ese resultado con cero (mediante el operador de relación de igualdad, representado en Java por el símbolo **==**). Si el resultado de la expresión lógica **numero % 2 == 0** es **true**, entonces se ejecutará el bloque primer bloque de sentencias (formado por una única sentencia dentro de las llaves) dando lugar a que se escriba por pantalla el mensaje **"El número es par."**.

3.2 ESTRUCTURA CONDICIONAL COMPUESTA: IF-ELSE.



Una vez que hemos visto en detalle la estructura condicional simple, vamos a estudiar ahora la estructura condicional compuesta, también conocida como **if**-ese. En este caso si tendremos que implementar los dos posibles caminos alternativos (bloques de sentencias), dependiendo de que la condición se evalúe como verdadera (**true**) o como falsa (**false**), tal y como se indica en la representación en diagrama de flujo de la imagen de la derecha:

- ✓ **Si la expresión que se evalúa es verdadera, se ejecuta la secuencia de instrucciones 1**.
- ✓ **Si es falsa, se ejecuta la secuencia de instrucciones 2**. En este caso, esta rama sí existe. Si no la hubiera, estaríamos en el caso anterior de una estructura condicional "simple".

En lenguaje natural (o pseudocódigo), esto se expresaría como:

Estructura condicional de doble alternativa. (Si-entonces-Si no).

```
Si expresion Entonces
   secuencia_1
Sino
   secuencia_2
Fin Si
```

Funcionamiento:
Si `expresion` es evaluada como verdadera, se ejecuta `secuencia_1` y en caso contrario, no se ejecuta `secuencia_1` y se ejecuta `secuencia_2`.

Fíjate que la palabra **Entonces** se pone para delimitar con claridad dónde termina la expresión que se va a evaluar y dónde empieza la secuencia de instrucciones del primer bloque. En la parte **Sino** no es necesario, ya que esa misma palabra delimita el final del primer bloque de instrucciones y el comienzo del segundo. Y se pondría un **Fin Si** que delimita dónde acaba la sentencia condicional, bien sea delimitando el final del único bloque de sentencias en el condicional simple o bien delimitando el segundo.

Una vez que tenemos clara la estructura, vamos a ver como se particulariza en el lenguaje Java, pues cada lenguaje de programación tendrá sus particularidades específicas.

La estructura condicional **if** puede presentarse en Java de las siguientes formas:

Estructura if de doble alternativa (o if-else).

Sintaxis para el caso de ejecución condicional de dos ramas y una sola sentencia en cada rama	Sintaxis para el caso de ejecución condicional de dos ramas y un bloque de sentencias en cada rama

`if (expresion_logica)`
`  sentencia_1;`
`else`
`  sentencia_2;`
	`if (expresion_logica) {`
	`  sentencia_1;`
	`  . . .`
	`  sentencia_N;`
	`} else {`
	`  sentencia_1;`
	`  . . .`
	`  sentencia_N;`
	`}`

Funcionamiento:
Si la evaluación de la `expresion_logica` ofrece un resultado verdadero, se ejecutará la primera sentencia o el primer bloque de sentencias. Si, por el contrario, la evaluación de la `expresion_logica` ofrece un resultado falso, no se ejecutará la primera sentencia o el primer bloque y sí se ejecutará la segunda sentencia o el segundo bloque.

Antes de seguir avanzando, veamos un ejemplo de uso: ¿cómo podríamos utilizar la estructura condicional **if-else** para completar el ejercicio del apartado anterior de manera que se muestre por pantalla un texto donde se indique si un número entero leído desde teclado es par o impar? (en el ejercicio anterior solo si indicaba si el número era par, pues no teníamos **else**).

Una forma muy sencilla para llevar a cabo la comprobación podría ser la siguiente:

```
1  if (numero % 2 == 0) {  // Si el resto de la división entera entre 2 es cero, el número es par
2      System.out.println("El número es par.");
3  } else { // Si no, el número es impar
4      System.out.println("El número es impar.");
5  }
```

Como puedes observar, se trata de añadir el componente **else** a la estructura de control condicional (líneas 3-5). Con esto podemos indicar qué queremos que se haga cuando:

1. el resultado de evaluar la condición sea **true**: **System.out.println("El número es par.");**
2. el resultado de evaluar la condición sea **false**: **System.out.println("El número es impar.");**

Ejercicio Resuelto

Teniendo en cuenta el ejercicio anterior, implementa el programa completo que lea desde teclado un número entero e indique si se trata de un número par o impar.

Se trataría simplemente de unir las dos partes:

1. La lectura del número;
2. La comprobación del número:

```
Scanner teclado = new Scanner(System.in);
int numero;
System.out.print("Introduzca un número entero: ");
numero = teclado.nextInt();
if (numero % 2 == 0) {  // Si el resto de la división entera entre 2 es cero, el número es par
    System.out.println("El número es par.");
} else { // Si no, el número es impar
    System.out.println("El número es impar.");
}
```

Algunas consideraciones más a tener en cuenta sobre las sentencias condicionales en Java:

- ✓ **La cláusula else de la sentencia if no es obligatoria**. En algunos casos no necesitaremos utilizarla, pero sí se recomienda cuando es necesario llevar a cabo alguna acción en el caso de que la expresión lógica no se cumpla.
- ✓ En aquellos casos en los que no existe cláusula **else**, si la expresión lógica es falsa, simplemente se continuarán ejecutando las siguientes sentencias que aparezcan bajo la estructura condicional **if** . Es lo que hemos llamado en el apartado anterior estructura condicional simple.
- ✓ **Los condicionales if e if-else pueden anidarse**, de tal forma que dentro de un bloque de sentencias puede incluirse otro **if** o **if-else**. El nivel de anidamiento queda a criterio del programador, pero si éste es demasiado profundo podría provocar problemas de eficiencia y legibilidad en el código. En otras ocasiones, un nivel de anidamiento excesivo puede hacernos pensar en la necesidad de utilización de otras estructuras de selección más adecuadas.

- ✓ Cuando se utiliza anidamiento de este tipo de estructuras, es necesario poner especial atención en saber a qué **if** está asociada una cláusula **else**. Normalmente, **un else estará asociado con el if inmediatamente superior o más cercano que exista dentro del mismo bloque y que no se encuentre ya asociado a otro else**.

Recomendación

Buena práctica de programación

- ✓ Utiliza la **sangría** o **indentación** en ambos cuerpos de una estructura **if-else**.
- ✓ Si hay **varios niveles de sangría**, en cada nivel debe aplicarse la **misma cantidad de espacios en blanco**.
- ✓ **Colocar siempre las llaves** en una instrucción **if-else** (y en general en cualquier estructura de control) ayuda a evitar que se omitan de manera accidental especialmente cuando posteriormente se agregan instrucciones a una cláusula **if** o **else**. Para evitar que esto suceda, algunos programadores prefieren escribir la llave inicial y final de los bloques antes de escribir las instrucciones individuales dentro de ellas.

Errores de programación

- ✓ **Olvidar una o las dos llaves que delimitan un bloque** puede provocar errores de sintaxis o errores lógicos en un programa.
- ✓ Colocar **un punto y coma después de la condición en una instrucción if-else** produce un error lógico en las instrucciones **if** de selección simple, y un error de sintaxis en las instrucciones **if-else** de selección doble (cuando la parte del **if** contiene una instrucción en el cuerpo).

¿Cuándo se mostrará por pantalla el mensaje incluido en el siguiente fragmento de código?

```
if (numero % 2 == 0);
    System.out.print("El número es par");
```

Siempre, ya que **al haber un punto y coma detrás de la expresión lógica de la estructura if, lo que se está diciendo es que si esa condición es verdadera, no se haga nada** (se haga un "punto y coma", que es la sentencia "nula" o "vacía"). Y por tanto la sentencia siguiente, que es la que imprime el mensaje en pantalla, estará ya fuera del bloque **if** y por tanto siempre se ejecutará.

Éste es un **error muy típico de programador "novato"** en el que escribimos un punto y coma justo después de la condición del **if** y a continuación, justo debajo, la sentencia (o bloque de sentencias) que deberían ejecutarse si la condición es verdadera. Sin embargo, al ejecutar observamos que esa sentencia se ejecuta siempre, independientemente de que la condición sea verdadera o falsa. Esto es porque la sentencia asociada al **if** es el punto y coma (sentencia nula o vacía) y el resto de sentencias que haya debajo están ya fuera del **if**. ¡Mucho cuidado con esto!

Para agravar la situación y que aún sea más difícil detectar este error, si hemos realizado una correcta indentación (sangrado hacia la derecha de las sentencias que pretendemos que estén dentro de una estructura de control) habremos escrito a la derecha la sentencia que está debajo del **if** (como de hecho sucede en este ejercicio), de manera que visualmente nuestro cerebro interpreta inmediatamente que esa sentencia solo se ejecutará si la condición es verdadera y en la mayoría de los casos no prestaremos atención al punto y coma. Observa que, aunque haya incluido un sangrado o indentación, en realidad es como si hubiera escrito lo siguiente:

```
if (numero % 2 == 0);  // Si el número es par, no hacer nada (el punto y coma es la sentencia "vacía")
System.out.print("El número es par"); // Esto se ejecuta siempre (tanto si la condición es true o false)
```

Ejercicio Resuelto

Nos han pedido que escribamos un programa en Java que solicite un número entero e indique por pantalla si se trata de un número positivo o no. Para ello se mostrará uno de estos dos posibles mensajes:

- ✓ "*El número es positivo*", si en efecto el número es positivo (mayor que cero).
- ✓ "*El número no es positivo*", si el número no es positivo (cero o negativo, es decir, menor o igual que cero).

Implementarlo utilizando una estructura condicional de tipo **if-else**.

La manera más sencilla de implementar este programa es simplemente construir una sentencia condicional con la condición **numero>0**. Si esa condición es verdadera (bloque de sentencias del **if**) se mostrará el mensaje de que el número es positivo, y si no es así (bloque de sentencias del **else**) se indicará que no es positivo:

```
Scanner teclado = new Scanner(System.in);
int numero;
System.out.print("Introduzca un número entero: ");
numero = teclado.nextInt(); // Lectura de un entero desde teclado
if (numero>0) {
    System.out.println("El número es positivo.");
} else {
    System.out.println("El número no es positivo.");
}
```

Dado que tanto el bloque **if** como el bloque **else** tan solo contienen una sentencia (una línea) podrían eliminarse las llaves de ambos bloques, quedando el código así:

```
int numero;
System.out.print("Introduzca un número entero: ");
numero = teclado.nextInt();
if (numero>0)
    System.out.println("El número es positivo.");
else
    System.out.println("El número no es positivo.");
```

Dependerá de ti si quieres usar las llaves o no en estos casos. Las llaves pueden aportar claridad, pero también es cierto que un exceso de llaves puede provocar lo contrario. Según vayas construyendo programas con una estructura más compleja lo irás descubriendo tú mismo.

Implementa ahora el programa utilizando la condición contraria que hayas usado antes. Lo más habitual es que hayas usado la condición `numero > 0`. Por tanto la condición contraría sería `numero <= 0` o bien `!(numero > 0)`.

En este caso, si lo que comprobamos es la condición inversa (que no sea positivo), en el bloque asociado al **if** habrá que colocar la frase "**El número no es positivo**" y la contraria en el bloque asociado al **else**.

Para ello podemos usar la condición **numero <= 0**:

```
System.out.print("Introduzca un número entero: ");
numero = teclado.nextInt();
if (numero<=0) {
    System.out.println("El número no es positivo.");
} else {
    System.out.println("El número es positivo.");
}
```

O bien la condición **!(numero > 0)**:

```
System.out.print("Introduzca un número entero: ");
numero = teclado.nextInt();
if (!(numero > 0)) {
    System.out.println("El número no es positivo.");
} else {
    System.out.println("El número es positivo.");
}
```

Tendría el mismo efecto.

Nuevamente aquí si lo deseas puedes evitar las llaves en todos aquellos bloques que sean de una única línea, que en este caso son todos.

¿Se te ocurre cómo podrías haber resuelto este mismo problema sin usar la estructura if-else? Como pista te decimos que podrías utilizar el operador condicional ? que ya vimos en la unidad anterior.

En este caso se trata de una condición tan sencilla que queda hasta más corto si se usa el operador condicional:

```
System.out.print("Introduzca un número entero: ");
numero = teclado.nextInt();
System.out.println ( numero > 0 ? "El número es positivo." : "El número no es positivo.");
```

Dado que una buena parte del texto se repite (lo único que cambia es que aparezca o no la palabra "no") podríamos hasta habernos ahorrado escribir dos veces todo el texto y decidir cuándo hay que incluir el "no". Por ejemplo:

```
System.out.print("Introduzca un número entero: ");
numero = teclado.nextInt();
String esPositivo= numero > 0 ? "" : "no "; // Si el número es positivo no se concatena nada; si no, concatenaremos "no"
System.out.println("El número " + esPositivo + "es positivo.");
```

Por último, ¿se te ocurre alguna manera de escribir el programa utilizando una estructura if simple sin else?

El problema lo podemos enfocar de muchas formas diferentes, pero quizá la más fácil sea hacer algo similar a lo que hemos hecho anteriormente con el operador condicional **?**. Usamos una cadena para almacenar el "no" o bien no almacenar nada, de manera que así generaremos el texto de resultado apropiado según cada caso. Por ejemplo, podríamos hacer algo así:

```
System.out.print("Introduzca un número entero: ");
numero = teclado.nextInt();
String esPositivo= ""; // Cadena vacía si es positivo o con el texto "no" si no lo es
if (numero <= 0) {
    esPositivo= "no ";
}
System.out.println("El número " + esPositivo + "es positivo.");
```

Esta sería una de las múltiples posibilidades. Se nos pueden ocurrir muchas formas diferentes de resolverlo. Como puedes observar, normalmente no hay una manera única de escribir un programa. Lo que sí hay que procurar es que sea claro y fácil de entender.

Dado que el bloque **if** tan solo contiene una sentencia (una línea), podríamos evitar en este caso las llaves:

```
System.out.print("Introduzca un número entero: ");
numero = teclado.nextInt();
String esPositivo= ""; // Cadena vacía si es positivo o con el texto "no" si no lo es
if (numero <= 0)
    esPositivo= "no ";
System.out.println("El número " + esPositivo + "es positivo.");
```

VER MÁS EJERCICIOS RESUELTOS EN LA PLATAFORMA

3.3 ESTRUCTURAS CONDICIONALES ANIDADAS.

Como ya hemos visto en algún ejemplo en el apartado anterior, **las estructuras de control condicionales (y en general cualquier estructura de control) pueden anidarse** de manera que el bloque de sentencias dentro de la parte **if** o de la parte **else** puede a su vez contener una nueva estructura **if-else**.

Veamos algunos ejemplos en los que pueda darse esa circunstancia.

1. Queremos mostrar por pantalla si un número entero x es **par y además superior a 100**. Para ello podríamos primero comprobar si es par y, a continuación, pero dentro del bloque de sentencias del **if**, comprobar si es superior a 100:

```
if ( x % 2 == 0) {
    if ( x > 100 ) {
        System.out.println ("El número es par y superior a 100.");
    }
}
```

El **if más interno** sólo contiene una sentencia dentro en su bloque, así podrían omitirse las llaves si así lo decidimos:

```
if ( x % 2 == 0 ) {
    if ( x > 100 )
        System.out.println ("El número es par y superior a 100.");
}
```

El **if más externo** también contiene una única sentencia en su bloque (la segunda sentencia **if**), de manera que también podríamos omitir sus llaves:

```
if ( x % 2 == 0 )
    if ( x > 100 )
        System.out.println ("El número es par y superior a 100.");
```

Ahora bien, en este caso la anidación no es necesaria, pues disponemos del operador lógico **AND** (**&&** o **&** en lenguaje Java) que nos permite unir ambas condiciones en una única codición más compleja

```
if ( x % 2 == 0 && x > 100 )
    System.out.println ("El número es par y superior a 100.");
```

Con esto vemos que no siempre es necesario el anidamiento y que el uso de operadores lógicos puede ayudarnos a simplicar la estructura de los programas.

2. Imaginemos que se nos pide ahora **comprobar si el número x es negativo, cero o positivo** indicando para cada caso un mensaje de texto apropiado. Para resolver este problema podríamos hacer una primera comprobación para ver si el número es negativo y, en caso contrario, como nos quedan dos posibles opciones (cero o positivo), hacer una nueva comprobación para ver si es positivo. Si no fuera positivo sabremos que se trata de la única alternativa que nos queda: cero. Estas comprobaciones podríamos implementarlas con un **if-else** que tendría un **else** cuyo bloque de sentencias contendría un nuevo **if-else**:

```
if (x < 0) {  // El número es negativo
    System.out.println("El número es negativo.");
} else {  // El número es cero o positivo
    if (x > 0 ) {  // El número es positivo
        System.out.println("El número es positivo.");
    } else {  // Si se ha llegado hasta aquí el número es obligatoriamente cero
        System.out.println("El número es cero.");
}
```

Nuevamente aquí podemos evitar todas las llaves, pues cada bloque de sentencias contiene una única sentencia (aunque alguna de ellas contenga a su vez nuevos bloques):

```
if (x < 0)
    System.out.println("El número es negativo.");
else
    if (x > 0 )
        System.out.println("El número es positivo.");
    else
        System.out.println("El número es cero.");
```

Ante esto podemos hacernos la pregunta: ¿nos conviene quitarlas todas o puede venirnos bien dejar algunas para mejorar la legibilidad del código? Esa percepción nos la irá proporcionando la experiencia según vayamos implementando cada vez programas más complejos.

Ejercicio Resuelto

Nos plantean la posibilidad de escribir un programa que a partir de una nota cuantitativa calcule su calificación cualtiativa equivalente teniendo en cuenta que:

- ✓ Si la nota es menor de 5, la calificación será **"INSUFICIENTE"**.
- ✓ Si la nota es mayor o igual a 5 y menor que 6, la calificación será **"SUFICIENTE"**.
- ✓ Si la nota es mayor o igual a 6 y menor que 7, la calificación será **"BIEN"**.
- ✓ Si la nota es mayor o igual a 7 y menor que 9, entonces la calificación será **"NOTABLE"**.
- ✓ Si la nota es mayor o igual a 9 y menor o igual a 10, entonces la calificación será de **"SOBRESALIENTE"**.

Escribe un programa en Java que a partir de una nota cuantitativa calcule su calificación cualtiativa equivalente utilizando una serie de estructuras **if** sin anidar.

Un primer acercamiento podría ser construir toda una serie de sentencias **if** comprobando cada uno de esos casos:

```
if ( notaCuantitativa < 5 )
    notaCualitativa= "INSUFICIENTE";
if ( notaCuantitativa >=5 && notaCuantitativa < 6  )
    notaCualitativa= "SUFICIENTE";
if ( notaCuantitativa >=6 && notaCuantitativa < 7  )
    notaCualitativa= "BIEN";
if ( notaCuantitativa >= 7 && notaCuantitativa < 9 )
    notaCualitativa= "NOTABLE";
if ( notaCuantitativa >=9 && notaCuantitativa<= 10 )
    notaCualitativa= "SOBRESALIENTE";

System.out.println ("La calificación cualitativa equivalente es: " + notaCualitativa);
```

Ahora bien, esto sería muy poco eficiente, pues si por ejemplo la nota es de un 4.2 la calificación debería ser "INSUFICIENTE" y no se deberían seguir haciendo comprobaciones. Sin embargo, tal y como lo hemos planteado aquí, siempre se van a llevar a cabo todas las comprobaciones (desde insuficiente hasta sobresaliente), aunque ya sepamos desde la primera comprobación que estamos en el rango de insuficiente.

Una vez que hemos resuelto el problema de esta primera forma y podemos constatar su escasa eficiencia, se nos plantea la posibilidad de utilizar **if** anidados de manera que sólo si la condición no se cumple (parte **else** de la estructura) entonces intentamos la siguiente. De esta manera, en cuanto se logre entrar en alguna de las condiciones, las demás ya no se realizarán, pues están dentro de un **else**.

Siguiendo esta nueva manera de plantear el problema escribe el nuevo código Java para ese programa.

Siguiendo esta nueva manera de plantear el problema nos quedaríamos con este código:

```
if ( notaCuantitativa < 5 ) {
    notaCualitativa= "INSUFICIENTE";
} else {
    if ( notaCuantitativa >=5 && notaCuantitativa < 6  ) {
        notaCualitativa= "SUFICIENTE";
    }
    else {
        if ( notaCuantitativa >=6 && notaCuantitativa < 7  ) {
            notaCualitativa= "BIEN";
        }
        else {
            if ( notaCuantitativa >= 7 && notaCuantitativa < 9 ) {
                notaCualitativa= "NOTABLE";
            }
            else {
                if ( notaCuantitativa >=9 && notaCuantitativa<= 10 ) {
                    notaCualitativa= "SOBRESALIENTE";
                }
            }
        }
    }
}

System.out.println ("La calificación cualitativa equivalente es: " + notaCualitativa);
```

Ahora bien, aquí hay condiciones que es innecesario comprobar, pues cada vez que se entra en un **else** es porque no se ha cumplido la condición de su **if**. Por ejemplo en el primer **else** ya no es necesario comprobar que **notaCuantitativa>=5**, pues si no ha sido menor que 5 es que en efecto es mayor o igual que 5. Y así con el resto de los **else** en los cuales no es necesario volver a comprobar condiciones que ya se han comprobado en sus respectivos **if**:

```
if ( notaCuantitativa < 5 ) {
    notaCualitativa= "INSUFICIENTE";
} else {  // Aquí ya sabemos que notaCuantitativa >=5. No es necesario comprobarlo
    if ( notaCuantitativa < 6  ) {
        notaCualitativa= "SUFICIENTE";
    }
    else {  // Aquí ya sabemos que notaCuantitativa >=6. No es necesario comprobarlo
        if ( notaCuantitativa < 7  ) {
```

```
            notaCualitativa= "BIEN";
        }
        else {  // Aquí ya sabemos que notaCuantitativa >=7. No es necesario comprobarlo
            if ( notaCuantitativa < 9 ) {
                notaCualitativa= "NOTABLE";
            }
            else {  // Aquí ya sabemos que notaCuantitativa >=9. No es necesario comprobarlo
                if ( notaCuantitativa<= 10 ) {
                    notaCualitativa= "SOBRESALIENTE";
                }
            }
        }
    }
}

System.out.println ("La calificación cualitativa equivalente es: " + notaCualitativa);
```

Si te fijas, tanto los **if** como los **else** de toda esa estructura anidada contienen **bloques de una sola sentencia**, de manera que podemos eliminar todas las llaves:

```
if ( notaCuantitativa < 5 )
    notaCualitativa= "INSUFICIENTE";
else
    if ( notaCuantitativa < 6  )
        notaCualitativa= "SUFICIENTE";
    else
        if ( notaCuantitativa < 7  )
            notaCualitativa= "BIEN";
        else
            if ( notaCuantitativa < 9 )
                notaCualitativa= "NOTABLE";
            else
                if ( notaCuantitativa<= 10 )
                    notaCualitativa= "SOBRESALIENTE";

System.out.println ("La calificación cualitativa equivalente es: " + notaCualitativa);
```

Ahora bien, es probable que tantos niveles de indentación tan seguidos perjudiquen a la legibilidad del código, haciéndolo difícil de seguir.

En estos casos con tanta indentación en estructuras donde se van encadenando los **else**, las "normas de estilo" de Java permiten y recomiendan que cuando haya un **if** (y nada más) dentro de un **else** puedan colocarse ambas palabras juntas (**else if**) y no incrementar otro nivel de indentación. De esta manera se consigue mejorar sensiblemente la **legibilidad**. De esta manera el código quedaría con el siguiente aspecto:

```
if ( condición_1 ) {
    bloque de sentencias 1
} else if ( condición_2  ) {
    bloque de sentencias 2
} else if ( condición_3  ) {
    bloque de sentencias 3
} ...
} else if ( condición_n  ) {
    bloque de sentencias n
} else {
    bloque de sentencias n+1
}
```

Lo que le proporciona un aspecto mucho más compacto y legible.

Escribe cómo quedaría nuestro programa utilizando este estilo más compacto.

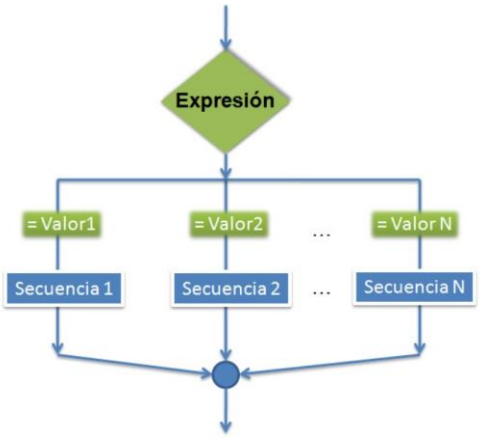
Nuestro programa de las calificaciones siguiendo ese modelo quedaría entonces así:

```
if ( notaCuantitativa < 5 )
    notaCualitativa= "INSUFICIENTE";
else if ( notaCuantitativa < 6  )
    notaCualitativa= "SUFICIENTE";
else if (  notaCuantitativa < 7  )
    notaCualitativa= "BIEN";
else if ( notaCuantitativa < 9 )
    notaCualitativa= "NOTABLE";
else if ( notaCuantitativa<= 10 )
    notaCualitativa= "SOBRESALIENTE";

System.out.println ("La calificación cualitativa equivalente es: " + notaCualitativa);
```

Lo cual parece bastante más sencillo de entender y de seguir, ¿no crees?

REALIZA EL EJERCICIO PROPUESTO EN LA PLATAFORMA DE ESTE MISMO APARTADO.

3.4 ESTRUCTURA SELECTIVA MÚLTIPLE: SWITCH.



¿Qué podemos hacer cuando nuestro programa debe elegir entre más de dos alternativas?

Una posible solución podría ser emplear estructuras **if** anidadas, aunque no siempre esta solución es la más eficiente, porque en caso de que las sucesivas condiciones que se van evaluando sean falsas, puede ser necesario comprobar todas antes de saber con seguridad en qué caso nos encontramos. ¿No sería deseable que con una sola comprobación pudiéramos ir directamente al caso apropiado y ejecutar las instrucciones que lleve asociadas?

Desde luego que sí, esto sería más eficiente, y en algunos casos es posible. Cuando estamos ante estas situaciones podemos utilizar la estructura de **selección múltiple,** que en Java es la sentencia **switch**, ya que al ejecutar el switch, se evalúa la expresión una única vez, y dependiendo del valor que tome, nos lleva a ejecutar el grupo de sentencias asociado, sin tener que volver a evaluar ninguna expresión más. Sin embargo, para conseguir eso a base de usar if/else anidados, cada caso debe ser comprobado, y si no se cumple, hay que seguir haciendo comprobaciones, evaluando las condiciones para decidir si se trata de cada caso particular. De esta forma, si el caso fuese el último de la lista, obliga a haber evaluado y comprobado todas las condiciones de los casos previos antes de descartarlos.

¿Cómo se implementa esta estructura en Java? En Java se la conoce como estructura switch. En la siguiente tabla se muestra tanto la sintaxis, como el funcionamiento de esta estructura en el lenguaje Java.

SINTAXIS	Condiciones

`switch (expresion){`
`  case valor1:`
`  sentencia1_1;`
`  sentencia1_2;`
`  . . .`
`  break;`
`  case valor2:`
`  sentencia2_1;`
`  sentencia2_2;`
`  . . .`
`  break;`
`  . . .`
`  case valorN:`
`  sentenciaN_1;`
`  sentenciaN_2;`
`  . . .`
`  break;`
`  default:`
`  sentencias_default;`
`}`
	✓ Donde expresión debe ser del tipo **char**, **byte**, **short** o **int**, y las constantes de cada **case** deben ser de este tipo o de un tipo compatible. Sólo desde la versión 7 del lenguaje Java se permiten expresiones de tipo **String** en la expresión, pero esto no funcionará con versiones anteriores del lenguaje.
	✓ La expresión debe ir entre paréntesis.
	✓ Cada **case** llevará asociado un valor y se finalizará con dos puntos (**:**).
	✓ El bloque de sentencias asociado a la cláusula **default** puede finalizar con una sentencia de ruptura **break** o no. No hay diferencia de funcionamiento. Se permite por mantener la estructura de las demás etiquetas, pero el compilador no lo necesita si el **default** va al final, tal y como se ve en el código, puesto que la llave de cierre delimita el final del **switch**.

Funcionamiento:

- Las diferentes alternativas de esta estructura estarán precedidas de la cláusula case que se ejecutará cuando el valor asociado al case coincida con el valor obtenido al evaluar la expresión del switch.
- Tradicionalmente, en las cláusulas case de Java **no podían indicarse expresiones condicionales, rangos de valores** o listas de valores (aunque otros lenguajes de programación sí lo permiten). Habrá que asociar una cláusula case a cada uno de los valores que deban ser tenidos en cuenta. Esto ha cambiado desde el JDK 12, que sí permite especificar listas de valores a través de expresiones lambda.
- Se pueden especificar varios case uno detrás de otro, de manera que para todos esos valores se ejecute la misma sección de código.
- La cláusula default será utilizada para indicar un caso por omisión o por defecto (cualquier otro caso no contemplado en las cláusulas case, en realidad). Las sentencias asociadas a la cláusula default se ejecutarán si ninguno de los valores indicados en las cláusulas case coincide con el resultado de la evaluación de la expresión de la estructura switch.
- La cláusula default puede no existir, y por tanto, si ningún case ha sido activado finalizaría el switch.
- Cada cláusula case puede llevar asociadas una o varias sentencias, sin necesidad de delimitar dichos bloques por medio de llaves.
- En el momento en el que el resultado de la evaluación de la expresión coincide con alguno de los valores asociados a las cláusulas case, se ejecutarán todas las instrucciones asociadas hasta la aparición de una sentencia break de ruptura (la sentencia break se analizará en epígrafes posteriores). Esto hace que si en una cláusula case no se incluye un break para finalizarla, al entrar en esa cláusula, se producirá un **"efecto en cascada",** de forma que seguirá ejecutando las sentencias de la siguiente cláusula case, y así sucesivamente las de todas las que sigan, hasta que en alguna encuentre una sentencia break, o hasta que alcance el final de la sentencia. Esto puede ser aprovechado para darle más versatilidad a la sentencia switch y hacer algo parecido a los rangos que sí permiten otros lenguajes, aunque sea de una forma "no muy limpia**".**

En resumen, se ha de comparar el valor de una expresión con un conjunto de valores constantes, si el valor de la expresión coincide con alguno de los valores constantes contemplados, se ejecutarán los bloques de instrucciones asociados al mismo. Si no existiese coincidencia, se podrán ejecutar una serie de instrucciones por omisión, si se ha incluido la cláusula **default**, o se saltará a la sentencia siguiente a esta estructura selectiva (tras la llave de cierre) si no se ha incluido esa cláusula.

Quizás esta estructura es la que más diferencias y particularidades presenta para distintos lenguajes de programación, de manera que si más adelante trabajas con otros lenguajes puede que te encuentres algo bastante distinto a la sintaxis de Java. Lo importante es que el planteamiento sigue siendo el mismo: se evalúa una expresión, y según su valor, se toma el camino que marca una de las etiquetas que identifican cada uno de los casos posibles.

Veamos un ejemplo sencillo. Imagina un programa que en función del valor de una variable entera llamada **switch** nos indique por pantalla el mensaje "cero", "uno", "dos" o bien "otro". Veamos cómo podría implementarse utilizando la estructura **switch** y también su equivalente mediante un grupo de estructuras **if-else** anidadas.

Ejemplo resuelto con switch	Ejemplo resuelto con if-else anidados

```
switch (numero) {
    case 0:
        System.out.println ("cero");
        break;
    case 1:
        System.out.println ("uno");
        break;
    case 2:
        System.out.println ("dos");
        break;
    default:
        System.out.println ("otro");
}
```

```
if (numero==1)
    System.out.println ("cero");
else if (numero==2)
    System.out.println ("uno");
else if (numero==3)
    System.out.println ("dos");
else
    System.out.println ("otro");
```

Ejercicio Resuelto

En el ejemplo anterior podríamos mejorar algunas cosas como por ejemplo disponer de una variable de tipo **String** que contenga el resultado final y de esa manera escribir una única línea de tipo **System.out.println** una vez finalizada la estructura de control, a modo de **salida de resultados**. También se podría haber realizado la solicitud de un número entero por teclado antes de comenzar (**entrada de datos**) y así ya tendríamos un programa completamente operativo.

Modifica y completa el programa del ejemplo anterior para que solicite un número entero por teclado y muestre el mensaje apropiado por pantalla utilizando la estructura:

1. entrada de datos:
2. procesamiento;
3. salida de resultados;

Para llevar a cabo este ejercicio e implementar un programa completamente operativo, lo primero que habría que hacer es declarar las dos variables que necesitamos: una de entrada con el número que debemos analizar y otra de salida con el texto de resultado que vamos a mostrar por pantalla.

```
// Declaración de variables
Scanner teclado = new Scanner(System.in);  // Recuerda que debes hacer import.util.Scanner al principio de todo
int numero;
String resultado;
```

Tras esto, habría que pedir por teclado un número entero (entrada de datos):

```
// Primera parte. Entrada de datos
System.out.print("Introduzca un número entero: ");
numero = teclado.nextInt();
```

Una vez que tengamos el dato de entrada, llevaríamos a cabo el análisis de ese dato para obtener un resultado (procesamiento). En este caso, en lugar de escribir inmediatamente en pantalla cuando averigüemos si es 0, 1, 2, etc. lo que hacemos es almacenar en la variable **resultado** el texto que queremos mostrar por pantalla sin llegar a escribirlo.

```
// Segunda parte. Procesamiento: calculamos un resultado a partir de los valores de entrada
switch (numero) {
    case 0:
        resultado= "cero";
        break;
    case 1:
        resultado= "uno";
        break;
    case 2:
        resultado= "dos";
        break;
    default:
        resultado= "otro";
}
```

Y finalmente mostraríamos por pantalla el resultado final (salida de resultados):

```
// Tercera parte. Salida de resultados
System.out.println (resultado);
```

El programa completo, juntándolo todo e incluyendo todos los elementos de nuestra plantilla "estándar" de programa, podría quedar más o menos así:

```
import java.util.Scanner;


public class EjemplosSwitch {


    public static void main(String[] args) {


        // Declaración de variables
        Scanner teclado = new Scanner(System.in);
        int numero;
        String resultado;


        // Entrada de datos
        System.out.print("Introduzca un número entero: ");
        numero = teclado.nextInt();


        // Procesamiento
        switch (numero) {
            case 0:
                resultado = "cero";
                break;
            case 1:
```

```
                resultado = "uno";
                break;
            case 2:
                resultado = "dos";
                break;
            default:
                resultado = "otro";
        }

        // Salida de resultados
        System.out.println("El número introducido es: " + resultado);

    }
}
```

En algunas ocasiones es posible que más de una opción de un **switch** dé lugar a que se tengan que realizar las mismas acciones. En tales casos lo que hay que hacer es acumular varias cláusulas **case** sin contenido una tras otra (o una debajo de otra) y un único cuerpo de sentencias para esas opciones. Algo así como:

```
switch (expresion) {
   case valor1:
   case valor2:
      sentencia1_1;  // Este bloque de sentencias se ejecutará si expresion da como resultado
tanto valor1 como valor2
      sentencia1_2;
      . . .
      break;

   case valor3:
   case valor4:
   case valor5:
      sentencia2_1;   // Aquí se entraría si resultado fuera valor3, valor4 o valor5
      sentencia2_2;
      . . .
      break;

   . . .
}
```

O bien si lo prefieres:

```
witch (expresion) {

   case valor1: case valor2:
      sentencia1_1;
      sentencia1_2;
      . . .
      break;

   case valor3:  case valor4:  case valor5:
      sentencia2_1;
      sentencia2_2;
      . . .
      break;

   . . .
}
```

Veamos un ejemplo donde podríamos utilizar esto. Imagina que dependiendo del valor de una cadena de caracteres llamada dia queremos mostrar por pantalla el siguiente mensaje:

- ✓ "día laborable", si el contenido es lunes, martes, miércoles, jueves o viernes;
- ✓ "fin de semana", si el contenido es sábado o domingo;
- ✓ "día no válido", en cualquier otro caso.

Este es un caso en el que nos vendrá muy bien aplicar lo anterior para no tener que repetir una y otra vez la misma línea de código.

```
switch (dia) {
    case "lunes":
    case "martes":
    case "miércoles":
    case "jueves":
    case "viernes":
        resultado = "día laborable";
        break;
    case "sábado":
    case "domingo":
        resultado = "fin de semana";
        break;
    default:
        resultado = "día no válido";
}
```

O bien, si prefieres escribir todos los **case** del mismo grupo en la misma línea:

```
switch (dia) {
    case "lunes":  case "martes":  case "miércoles":  case "jueves":  case "viernes":
        resultado = "día laborable";
        break;
    case "sábado":  case "domingo":
        resultado = "fin de semana";
        break;
    default:
        resultado = "día no válido";
}
```

Ten en cuenta que para el compilador de Java, tanto el espacio en blanco como el salto de línea realizan la función de separador, de manera que puedes usar espacios o bien saltos de línea para separar cada **case**. Tan solo afectará al aspecto visual, pero no a su significado o funcionamiento.

El código equivalente utilizando estructuras de tipo **if-else** habría sigo el siguiente:

```
if (dia.equals("lunes") || dia.equals("martes") || dia.equals("miércoles") || dia.equals("jueves")
|| dia.equals("viernes"))
    resultado = "día laborable";
else if (dia.equals("sábado") || dia.equals("domingo"))
    resultado = "fin de semana";
else
    resultado = "día no válido";
```

Fíjate que la acumulación de varios case para una única acción común es equivalente al uso de la estructura **OR** (en Java simbolizada por **||** o **|**) en la estructura if-else.

Recuerda que en Java la comparación entre cadenas de caracteres debe hacerse utilizando **.equals**, pues **String no es un tipo primitivo sino una clase** (fíjate que se escribe con la primera letra en mayúscula y no todo en minúsculas como **int**, **char**, **double**, etc.) y por tanto no podemos usar **==**. El motivo ya lo veremos más adelante. Pero recuerda siempre que si quieres comparar referencias a objetos que son instancias de una clase no puedes usar == sino **equals** o cualquier otro método de comparación que esa clase proporcione. Por ahora solo tendremos trabajar con referencias a objetos que sean instancias de la clase **String**. En la siguiente unidad, cuando empecemos a instanciar objetos y utilizarlos, profundizaremos en este tema.

Error común de programación: Olvidar una instrucción **break** cuando se necesita en una instrucción **switch** es un error lógico. Es decir, un error del que no nos va a avisar el compilador, pero que es muy probable que de lugar a un comportamiento no deseado de nuestro programa.

Ejercicio Propuesto

Recuerda que **para leer de teclado una cadena de caracteres** podemos hacer algo como:

```
Scanner teclado = new Scanner(System.in);
String dia;


System.out.print("Introduzca un día de la semana: ");
dia = teclado.nextLine();
```

Es decir, que utilizamos **nextLine** en lugar de **nextInt** y obviamente asignamos lo que se obtenga a una variable de tipo **String** y no **int**.

Teniendo en cuenta lo anterior, completa el código del último ejemplo para escribir un programa completo que pueda ejecutarse y lleve a cabo la función de escribir por pantalla el mensaje apropiado (*día laborable*, *fin de semana*, *día no válido*) en función de la entrada recibida (*lunes*, *martes*, *miércoles*, etc.).

Ejercicio Resuelto

Realiza un programa en Java que calcule la nota de un examen de tipo test de 20 preguntas, donde ha habido 17 aciertos, 3 errores y 0 preguntas sin contestar, siguiendo la formula explicada en apartados anteriores. **La nota calculada debes obtenerla como un número entero, aunque los cálculos los puedes hacer con números reales.** Después de calcular la nota final, haz que el programa muestre la calificación no numérica de dicho examen:

- Si la nota es 0, 1, 2, 3 o 4, la calificación será **"INSUFICIENTE"**.
- Si la nota es 5, la calificación será **"SUFICIENTE"**.
- Si la nota es 6, la calificación será **"BIEN"**.
- Si la nota es 7 o 8, entonces la calificación será **"NOTABLE"**.
- Si la nota es 9 o 10, entonces la califación será de **"SOBRESALIENTE"**.

Para realizar este ejercicio deberás usar obligatoriamente la estructura switch.

```
package ut02;

/**
 * Ejemplo de uso del switch.
 */
public class CondicionalSwitch {

    /**
     * Vamos a realizar el cálculo de la nota de un examen de tipo test. Para
     * ello, tendremos en cuenta el número total de preguntas, los aciertos y
     * los errores. Dos errores anulan una respuesta correcta. La nota que vamos
     * a obtener será un número entero. Finalmente, se muestra por pantalla la
     * nota obtenida, así como su calificación no numérica. La obtención de
     * la calificación no numérica se ha realizado utilizando la estructura
     * condicional múltiple o switch.
     */
    public static void main(String[] args) {
```

```
        // Declaración e inicialización de variables
        int numeroAciertos = 17;
        int numeroErrores = 3;
        int numeroPreguntas = 20;
        float nota = 0;
        int notaEntera = 0;
        String calificacion="";

        /* Paso 1: Calculo de la nota flotante.
           Nota: "2f" es un literal de número flotante, al introducirlo,
                el conjunto de las operaciones se realizan con decimales. */
        nota = ((numeroAciertos - (numeroErrores/2f))*10f)/numeroPreguntas;

        /* Paso 2: Convertimos la nota flotante a entero.
        Para esta conversión, usamos el método round de la clase Math cuyo
        objetivo es precisamente redondear un número flotante a entero. */

        notaEntera=Math.round(nota);

        /* Nota: la conversión de float a int se podría haber hecho así:
                notaEntera=(int)nota;
           Pero la nota se hubiese truncado (eliminado la parte decimal),
           con lo que 8.2 sería 8 y 8.8 sería 8 también.*/


        /* Paso 3: Calculamos cual sería su calificación no numérica. */
        switch (notaEntera) {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
                calificacion = "INSUFICIENTE";
                break;
            case 5:
                calificacion = "SUFICIENTE";
               break;
            case 6:
               calificacion = "BIEN";
               break;
            case 7:
            case 8:
                calificacion = "NOTABLE";
               break;
            case 9:
            case 10:
               calificacion = "SOBRESALIENTE";
               break;
            default:
                System.out.println("Se ha introducido una nota errónea");
               calificacion = "CALIFICACIÓN FUERA DE RANGO";
        }

        /* Paso 4: Mostramos la calificación obtenida y su equivalente no
        numérico. */
        System.out.println ("La nota obtenida es: " + nota);
        System.out.println ("Y la calificación obtenida es: " + calificacion);
    }
}
```

4. ESTRUCTURAS REPETITIVAS, ITERATIVAS O CÍCLICAS.

Nuestros programas ya son capaces de controlar su ejecución teniendo en cuenta determinadas condiciones, pero aún hemos de aprender un conjunto de estructuras que nos permita **repetir una secuencia de instrucciones** determinada hasta que se consiga un determinado objetivo o hasta que se cumpla una determinada condición.

La función de estas **estructuras repetitivas** es repetir la ejecución de una serie de instrucciones teniendo en cuenta una condición.

A este tipo de estructuras se las denomina **estructuras de repetición**, estructuras repetitivas, **bucles** o estructuras iterativas. En Java existen cuatro clases de bucles:

- ✓ Bucle **while** (repite mientras).
- ✓ Bucle **do-While** (repite hasta).
- ✓ Bucle **for** (repite para).
- ✓ Bucle **for/in** (repite para cada).

Los bucles **for** y **for/in** se consideran bucles **controlados por contador.** Por el contrario, los bucles **while** y **do-while** se consideran bucles **controlados por sucesos.**

La utilización de unos bucles u otros para solucionar un problema dependerá en gran medida de las siguientes preguntas:

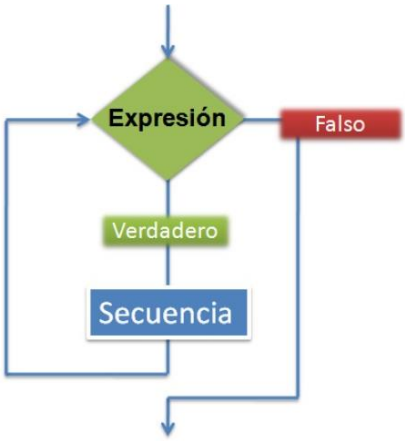
- ✓ ¿Sabemos a priori cuántas veces necesitamos repetir un conjunto de instrucciones?
- ✓ ¿Sabemos si hemos de repetir un conjunto de instrucciones si una condición satisface un conjunto de valores?
- ✓ ¿Sabemos hasta cuándo debemos estar repitiendo un conjunto de instrucciones?
- ✓ ¿Sabemos si hemos de estar repitiendo un conjunto de instrucciones mientras se cumpla una condición?
- ✓ ¿Sabemos si esas instrucciones se deben ejecutar siempre, al menos una primera vez, con independencia del resultado de evaluar la condición que controla el bucle?

Estas y otras preguntas tendrán su respuesta en cuanto analicemos cada una de las estructuras repetitivas en detalle.

Recomendación

Estudia cada tipo de estructura repetitiva, conoce su funcionamiento y podrás llegar a la conclusión de que **algunos de estos bucles son equivalentes entre sí**. Un mismo problema, podrá ser resuelto empleando diferentes tipos de bucles y obtener los mismos resultados. De hecho, sería posible resolver cualquier problema si sólo contáramos con una estructura de control repetitiva (un único tipo de bucle), pudiendo ser cualquiera de ellos. No obstante, según el tipo de problema, disponer de varios tipos podrá permitirnos construir soluciones más simples y claras, lo que siempre resulta muy deseable.

4.1 ESTRUCTURA REPETITIVA WHILE.



El bucle **while** es la primera estructura **de repetición controlada por sucesos** que vamos a estudiar. La utilización de este bucle responde al planteamiento de la siguiente pregunta:

¿Qué podemos hacer si lo único que sabemos es que se han de repetir un conjunto de instrucciones mientras se cumpla una determinada condición?

La característica fundamental de este tipo de estructura repetitiva estriba en ser **útil en aquellos casos en los que las instrucciones que forman el cuerpo del bucle pod ría ser necesario ejecutarlas o no**.

Es decir, en el bucle **while** siempre se evaluará la condición que lo controla, y si dicha condición es cierta, el cuerpo del bucle se ejecutará una vez, y se seguirá ejecutando mientras la condición sea cierta. Pero si en la evaluación inicial de la condición ésta no es verdadera, el cuerpo del bucle no se ejecutará.

Es imprescindible que en el interior del bucle **while** se realice alguna acción que modifique la condición que controla la ejecución del mismo, en caso contrario estaríamos ante un bucle infinito, que es algo que se debe evitar siempre.

¿Cómo se implementa esta estructura en Java? En la siguiente tabla se muestra tanto la sintaxis como el funcionamiento de esta estructura (ciclo **while**) en el lenguaje Java.

Sintaxis para el caso de una sola sentencia en el cuerpo del bucle while	**Sintaxis para el caso de un bloque de sentencias en el cuerpo del bucle while**

`while (condición)`
`  sentencia;`
	`while (condición) {`
	`  sentencia_1;`
	`  . . .`
	`  sentencia_n;`
	`}`

Funcionamiento: • Mientras la condición sea cierta, el bucle se repetirá, ejecutando las instrucciones de su interior (una o varias). En el momento en el que la condición no se cumpla, el control del flujo del programa pasará a la siguiente instrucción que exista justo detrás del bucle while. • La condición se evaluará siempre al principio, y podrá darse el caso de que las instrucciones contenidas en él no lleguen a ejecutarse nunca si no se satisface la condición de partida.	

Veamos un ejemplo. Imagina un programa que queremos que muestre por pantalla los números del 1 al 5. Inicialmente podríamos pensar en escribir cinco sentencias de escritura por pantalla:

```
System.out.println (1);
System.out.println (2);
System.out.println (3);
System.out.println (4);
System.out.println (5);
```

Pero claro, basta pensar que si hubiéramos tenido que escribir los números del 1 al 500 no parece una solución nada razonable tener que escribir quinientas veces esas líneas en nuestro programa. Precisamente para esto sirven los bucles. Podríamos declarar una variable de tipo entero que fuera almacenando el número que queremos mostrar en pantalla e implementar un bucle cuya condición de "continuidad" fuera "*mientras el número esté por debajo de 5, incluido el propio 5*". A esa variable de tipo entero podríamos darle el valor inicial 1 y podríamos ir incrementándola en una unidad dentro del cuerpo del bucle. De este modo, cuando superara el valor 5 (y llegara a valer 6) ya no se entraría en el cuerpo del bucle (la evaluación de la condición sería **false**) y el programa continuaría justo por la siguiente sentencia después del cuerpo del bucle.

Veámoslo paso a paso:

1.- Inicialización de la variable entera que iremos mostrando por pantalla:

```
int numero=1;
```

2.- Cabecera de bucle while con su comprobación: "*mientras el número sea menor o igual que cinco*":

```
while ( numero<=5 )
```

3.- Cuerpo del bucle while donde se muestre el valor de la variable y a continuación se incremente en uno:

```
{
    System.out.println (numero);
    numero++;
}
```

Dado que el cuerpo tiene más de una sentencia necesitas escribir el bloque entre llaves. Si tan solo hubiera una línea podrías haber omitido las llaves. Por ejemplo, si hubieras usado el operador "post-incremento" dentro del propio **println**:

```
System.out.println (numero++);  // Esta línea es equivalente a las dos líneas anteriores y ya no son necesrias las llaves de bloque
```

Uniéndolo todo nos podría quedar algo como:

```
int numero=1;
while ( numero<=5 ) {
    System.out.println (numero);
    numero++;
}
```

O bien así (si lo hacemos todo en una única línea):

```
int numero=1;
while ( numero<=5 )
    System.out.println (numero++);
```

Y el resultado que obtendríamos por pantalla quedaría de la siguiente manera:

```
1
2
3
4
5
```

Ejercicio resuelto

Escribe un programa que solicite dos números por teclado (inicio y fin, donde inicio debería ser menor o igual que fin) y muestre por pantalla todos los números que van desde inicio hasta fin, todos en una misma línea.

Aquí tienes un ejemplo de una posible ejecución del programa:

```
Introduzca el inicio: 4
Introduzca el fin: 12


Secuencia de números desde 4 hasta 12
4 5 6 7 8 9 10 11 12
```

Aquí tienes una muestra de cómo podría quedar tu programa:

```
// Declaración de variables
Scanner teclado = new Scanner(System.in);
int inicio, fin, numero;

// Entrada de datos
System.out.print ("Introduzca el inicio: ");
inicio = teclado.nextInt();

System.out.print("Introduzca el fin: ");
fin = teclado.nextInt();

// Procesamiento y salida de resultados
System.out.println ("\nSecuencia de números desde " + inicio + " hasta " + fin);
numero= inicio;
while (numero <= fin) {
    System.out.print (numero + " ");
    numero++;
}
System.out.println ();
```

O bien, si usas el operador de post-incremento en la misma línea de escritura por pantalla:

```
while (numero <= fin)
    System.out.print (numero++ + " ");
```

Recuerda que en este caso te interesa usar el operador post-incremento pues primero queremos que se muestre el valor de la variable por pantalla y después se realice el incremento. Si hubiera utilizado el operador pre-incremento haciendo lo siguiente:

```
while (numero <= fin)
    System.out.print (++numero + " ");
```

¿Qué crees que sucedería?

Una utilidad muy interesante de las estructuras repetitivas es la posibilidad de volver a pedir un dato de entrada si este no cumple alguna condición.
Por ejemplo, imagina que nos piden unos números de inicio y de fin y que el fin no pueda ser mayor que el inicio. Podríamos realizar una primera lectura de los valores:

```
// Entrada de datos
System.out.println ("Debe introducir el incio y el fin.");
System.out.println ("Tenga en cuenta que fin no debe ser menor que el inicio: ");
System.out.print ("Introduzca el inicio: ");
inicio = teclado.nextInt();
System.out.print ("Introduzca el fin: ");
fin = teclado.nextInt();
```

Y a continuación comprobar mediante la condición de un **while** si los valores cumplen o no la condición que deben cumplir:

```
while (inicio > fin)  // Si inicio es superior a fin hay que volver a solicitar
```

En caso de que se cumpla la condición de que inicio es mayor que fin (que es lo que no debe suceder), se entraría en el cuerpo del **while** y se volvería a realizar la petición:

```
// Si inicio es superior a fin hay que volver a solicitar
{
    System.out.println ("Error: el fin no puede ser superior al inicio.");
    System.out.print ("Introduzca el inicio: ");
    inicio = teclado.nextInt();
    System.out.print ("Introduzca el fin: ");
    fin = teclado.nextInt();
}
```

Y como está dentro de un **while** se volverá a comprobar una y otra vez hasta que introduzcamos los valores correctamente.
Uniéndolo todo nos quedaría algo así:

```
// Entrada de datos
System.out.println ("Debe introducir el incio y el fin.");
System.out.println ("Tenga en cuenta que el fin no debe ser menor que el inicio: ");
System.out.print ("Introduzca el inicio: ");
inicio = teclado.nextInt();
System.out.print ("Introduzca el fin: ");
fin = teclado.nextInt();

while (inicio > fin) { // Si inicio es superior a fin hay que volver a solicitar
    System.out.println ("Error: el fin no puede ser superior al inicio.");
    System.out.print ("Introduzca el inicio: ");
    inicio = teclado.nextInt();
    System.out.print ("Introduzca el fin: ");
    fin = teclado.nextInt();
}
System.out.println ("Entrada correcta.");
```

De esta manera podríamos obligar al usuario a introducir unos datos de entrada válidos y mientras no lo haga no se podrá avanzar en el programa y se seguirán pidiendo los valores indefinidamente. Fíjate que *while* en inglés significa precisamente "mientras".

Aquí tienes una muestra de cómo podría quedar su funcionamiento:

```
Debe introducir el incio y el fin.
Tenga en cuenta que el fin no debe ser menor que el inicio:
Introduzca el inicio: 5
Introduzca el fin: 2
Error: el fin no puede ser superior al inicio.
Introduzca el inicio: 8
Introduzca el fin: 5
Error: el fin no puede ser superior al inicio.
Introduzca el inicio: 2
Introduzca el fin: 6
Entrada correcta.
```

Una pega que podemos encontrar a esta manera de hacer una comprobación de validez de entradas es que tenemos que escribir dos veces la solicitud de los valores. Una primera vez antes del bucle y luego una segunda vez dentro del cuerpo del bucle, lo cual no parece muy práctico. Esto lo podremos resolver con la estructura **do-while**, que veremos en el siguiente apartado.

Error de programación: Si en el cuerpo de una instrucción **while** no se proporciona una acción que ocasione que en algún momento la condición de un while no se cumpla, por lo general se producirá un error lógico conocido como ciclo infinito, en el que el ciclo nunca terminará.

Ejercicio Propuesto

Escribe en Java un programa que solicite un **número n** para calcular la **tabla de multiplicar de ese número n** usando un bucle tipo **while**.

Autoevaluación

Utilizando el siguiente fragmento de código estamos construyendo un bucle infinito.

```
while (true) System.out.println("Imprimiendo desde dentro del bucle...");
```

La condición del bucle siempre será cierta (recuerda que un literal booleano es en sí mismo una expresión booleana, así que el literal **true** es una expresión que se evalúa siempre como **true**) y estará continuamente ejecutándose. ¡¡Hay que evitar el uso de este tipo de condiciones!!

Normalmente sólo reflejan una mala construcción de la lógica del programa, porque para no caer en bucles infinitos obligan a incluir sentencias de salto incondicional en el interior del bucle, que se ejecutan dentro de una estructura condicional que comprueba el momento en que debe salirse del bucle. ¡¡¡Para eso está la condición del bucle, para comprobar ahí cuál es la condición de salida!!

La salida natural de un bucle es tras comprobar la condición de salida para ver si hay que volver a hacer otra iteración o no, y todo lo que sea esquivar esa forma de hacer las cosas, es generar código poco claro, difícil de entender y de mantener, lo que redunda en un mayor coste.

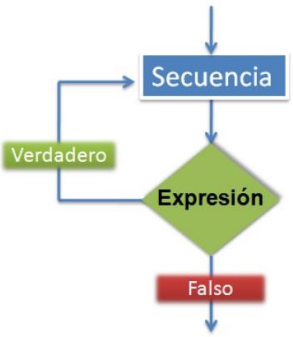
Ejercicio Resuelto

Escribe en Java un programa para calcular la **tabla de multiplicar del 7** usando un bucle tipo **while**.

Una solución posible sería:

```
/**
 * Mostrar la tabla de 7 usando una estructura repetitiva while.
 * @author Profesor
 */
public class RepetitivaWhile {
    /**
    * En esta solución se utiliza la estructura repetitiva while para representar
    * en pantalla la tabla de multiplicar del siete.
    */
    public static void main(String[] args) {
       // Declaración e inicialización de variables
       int numero = 7;
       int contador;
       int resultado=0;

       /* Paso 1. Mostrar la cabecera de la tabla */
       System.out.println ("Tabla de multiplicar del " + numero);
       System.out.println (".............................. ");

       /* Paso 2. Calcular la tabla de multiplicar del 7 usando un bucle while.
        * En este caso la inicialización de la variable contador hay que hacerla
        * antes de entrar en el bucle y la actualización del contador se realizaría
        * dentro del bucle.
        */

       contador = 1; //inicializamos la variable contadora
       while (contador <= 10){ //Establecemos la condición del bucle
           resultado = contador * numero;
           System.out.println(numero + " x " + contador + " = " + resultado);

           //Modificamos el valor de la variable contadora, para hacer que el
           //bucle pueda seguir iterando hasta llegar a finalizar.
           contador++;
       }
   }
}
```

4.2 ESTRUCTURA REPETITIVA DO-WHILE.



La estructura **do-while** es otro tipo de estructura **repetitiva controlada por sucesos.** En este caso, la pregunta que nos planteamos es la siguiente:

¿Qué podemos hacer si sabemos que se han de ejecutar un conjunto de instrucciones **al menos una vez**, y que dependiendo del resultado, puede que deban seguir repitiéndose mientras que se cumpla una determinada condición?

La característica fundamental de este tipo de estructura repetitiva estriba en ser útil en aquellos casos en los que las instrucciones que forman el cuerpo del bucle necesitan ser ejecutadas **al menos una vez** y repetir su ejecución mientras que la condición sea verdadera. Por tanto, en esta estructura repetitiva se ejecuta el cuerpo del bucle **siempre una primera vez**.

Como en el caso de **while**, para **do-while** también es imprescindible que en el interior del bucle se realice alguna acción que modifique la condición que controla la ejecución del mismo, en caso contrario estaríamos ante un bucle infinito.
¿Cómo se implementa esta estructura en Java? En la siguiente tabla se muestra tanto la sintaxis como el funcionamiento de esta estructura (ciclo **wo-hile**) en el lenguaje Java.

Sintaxis para el caso de una sola sentencia en el cuerpo del bucle do-while	**Sintaxis para el caso de un bloque de sentencias en el cuerpo del bucle do-while**

`do`
`  sentencia;`
`while (condición);`
	`do {`
	`  sentencia_1;`
	`  . . .`
	`  sentencia_N;`
	`} while (condición);`

Funcionamiento:

- El cuerpo del bucle se ejecuta la primera vez, a continuación, se evaluará la condición y si ésta es verdadera, el cuerpo del bucle volverá a repetirse, y así sucesivamente, hasta que la condición sea falsa.
- El bucle finalizará cuando la evaluación de la condición sea falsa, por tanto. En ese momento el control del flujo del programa pasará a la siguiente instrucción que exista justo detrás del bucle do-while.
- La condición se evaluará siempre después de una primera ejecución del cuerpo del bucle, por lo que no se dará el caso de que las instrucciones contenidas en él no lleguen a ejecutarse nunca.

Resumidamente:

- "Primero dispara y luego pregunta".

Ahora podemos repetir el ejemplo que hicimos para mostrar los números del 1 al 5 con la estructura **while** utilizando esta nueva estructura **do-while**. No habrá mucha diferencia:

```
int numero=1;
do { // El cuerpo del bucle siempre se ejecuta al menos la primera vez
    System.out.println (numero);
    numero++;
} while ( numero<=5 ); // Dependiendo de la evaluación de la condición se volverá a ejecutar o no
```

O bien, si el cuerpo tiene una única línea, podemos eliminar las llaves de bloque:

```
numero=1;
do
    System.out.println (numero++);
while ( numero<=5 );
```

Ejercicio Resuelto

Escribe un programa que solicite dos números por teclado (inicio y fin) y muestre por pantalla todos los números que van desde inicio hasta fin, todos en una misma línea. Este ejercicio ya lo hemos planteado usando un bucle **while**. En este caso debes utilizar un bucle **do-while**.

La solución es muy similar:

```
// Declaración de variables
Scanner teclado = new Scanner(System.in);
int inicio, fin, numero;

// Entrada de datos
System.out.print ("Introduzca el inicio: ");
inicio = teclado.nextInt();
System.out.print("Introduzca el fin: ");
fin = teclado.nextInt();

// Procesamiento y salida de resultados
System.out.println ("\nSecuencia de números desde " + inicio + " hasta " + fin);
numero= inicio;
do {
    System.out.print (numero + " ");
    numero++;
} while (numero <= fin);
System.out.println ();
```

Si retomamos el ejemplo en el que se comprobaba la validez de los datos de entrada utilizando un bucle **while**, recordarás la pega que le poníamos de tener obligatoriamente que escribir dos veces las sentencias de lectura. Ese problema desaparece con la estructura **do-while**, pues el cuerpo del bucle se va a ejecutar al menos una vez, ya que la comprobación no se hará hasta el final. En tal caso la entrada de datos con comprobación de validez nos podría quedar mucho más sencilla y compacta:

```
System.out.println ("Debe introducir el incio y el fin.");
do { // La primera vez siempre se entra en el cuerpo del bucle
    System.out.println ("Tenga en cuenta el fin que no debe ser menor que el inicio: ");
    System.out.print ("Introduzca el inicio: ");
    inicio = teclado.nextInt();
    System.out.print ("Introduzca el fin: ");
    fin = teclado.nextInt();
} while (inicio > fin);  // Si la condición no se cumple, se vuelve a ejecutar el cuerpo del bucle
System.out.println ("Entrada correcta.");
```

De esta manera evitamos tener que escribir una lectura de valores inicial antes de bucle y luego otra lectura exactamente igual en el cuerpo del bucle, por si hay que volver a introducir los valores una segunda vez (o una tercera, o una cuarta).

Ejercicio Resuelto

Escribe un programa que solicite un número par, de tal modo que si no se introduce un número par vuelva a solicitarlo hasta que así sea.

Bastaría con incluir la solicitud del número dentro de un bucle **do-while** de manera que mientras el número introducido no sea par (divisible entre dos) se siga ejecutando el bucle una y otra vez:

```
// Declaración de variables
int numero;


// Entrada de datos con validación
do {
    System.out.print ("Introduzca un número par: ");
    numero = teclado.nextInt();
} while ( numero % 2 != 0 ); // Mientras el número no sea divisible entre 2 se repite el bucle
System.out.println ("Entrada válida. Número " + numero);
```

Ejercicio Resuelto

Al igual que se pedía en apartados anteriores, ahora vamos os vamos a pedir que realiceís el ejercicio de la tabla de multiplicar del número 7, pero usando un bucle **do-while**.

Una solución posible es la siguiente:

```
/**
 * Mostrar la tabla de 7 usando una estructura repetitiva do-while.
 *
 * @author Profesor
 */
public class RepetitivaTipoDoWhile {


    /**
     * En esta solución se utiliza la estructura repetitiva while para
     * representar en pantalla la tabla de multiplicar del siete.
     */
    public static void main(String[] args) {

        // Declaración e inicialización de variables
        int numero = 7;
        int contador;
        int resultado = 0;

        /* Paso 1. Mostrar la cabecera de la tabla */
        System.out.println("Tabla de multiplicar del " + numero);
        System.out.println("............................ ");
```

```
        /* Paso 2. Calcular la tabla de multiplicar del 7 usando un bucle
         * do-while.
         * Al igual que ocurría en el bucle do-while, la variable contador
         * hay que inicializarla antes de entrar en el bucle, pero hay que tener
         * en cuenta que el bucle se ejecutará al menos una vez.
         * Después, la variable contador se modifica dentro del bucle para que
         * el bucle termine en algún momento y no se convierta en un bucle
         * infinito.
         */

        contador = 1; //Inicializamos la variable contador antes de empezar el bucle
        do {
            resultado = contador * numero;
            System.out.println(numero + " x " + contador + " = " + resultado);
            //Modificamos el valor de la variable contadora, para hacer que el
            //bucle pueda seguir iterando hasta llegar a finalizar (no hasta el infinito)
            contador++;
        } while (contador <= 10); //Establecemos la condición del bucle


    }
}
```

Recomendación: Buena práctica de programación

Incluye siempre llaves en una instrucción **do-while**, aún cuando estas no sean necesarias. Esto ayuda a eliminar ambigüedad entre las instrucciones **while** y **do-while** que contienen una sola instrucción.

Autoevaluación

En las sentencias do-while, ¿qué hay que tener siempre presente?

La necesidad de que dentro del cuerpo del bucle se incluya alguna sentencia que modifique la variable de control del ciclo, de forma que pueda verse alterado el valor de verdad de la condición que controla el bucle de tal manera que garanticemos que en algún momento se alcance la condición de salida, sin entrar en un bucle infinito. No basta con una sentencia que modifique la variable de control del ciclo, ya que si ello no implica que la condición de salida se alcance en algún momento, estaríamos igualmente en un bucle infinito.

Un bucle tipo **do-while**, que no contenga en su cuerpo ninguna sentencia capaz de modificar el valor de verdad de la condición que controla el ciclo, o bien se ejecuta una sola vez, o bien entra en un bucle infinito ejecutándose indefinidamente:
Es verdadero porque si no hay nada dentro del bucle que modifique el valor de la condición, cuando lleguemos a evaluarla ya habremos ejecutado las sentencias del mismo una vez, y pueden pasar dos cosas:

1. Que la condición sea falsa, en cuyo caso hemos alcanzado la condición de salida, y no se ejecutará ninguna vez más (pero ya se había ejecutado una).
2. Que la condición sea verdadera, en cuyo caso volverá ejecutarse de nuevo, pero como nada hay en esa iteración que modifique el valor de la condición, al final de cada iteración siempre vamos a encontrarnos con que sigue siendo verdadera, por lo que seguiremos repitiendo iteración tras iteración. ¡¡hasta el infinito y más allá!!

4.3 CONCEPTO DE CONTADOR.

En muchas ocasiones cuando se implementa un bucle, suele disponer de una variable que se va incrementando (o decrementando) a medida que se va realizando iteraciones sobre ese bucle. A este tipo de variables se les suele llamar **contadores**.

En algunos de los ejemplos que ya hemos visto en apartados anteriores aparecían ese tipo de variables "contadoras":

- ✓ para ir desde 1 hasta 5, en el primer ejemplo que se planteó. Variable **numero**;
- ✓ para ir desde inicio hasta fin, en el ejemplo de la secuencia de números. Variable **numero**;
- ✓ para ir desde 1 hasta 10, en las tablas de multiplicar. Variable **contador**.

Ahora bien, no siempre tiene por qué haber un contador asociado a un bucle. Por ejemplo en el caso de la comprobación de validez de entradas no se utilizaba ninguna variable para saber cuántas veces se había tenido que repetir la entrada de datos. No se hacía porque no se ha considerado útil o necesario. Si se hubiera considerado así podría haberse hecho sin problema.

Veamos un ejemplo más de una variable de tipo contador que pueda resultarnos útil. Imagina un cajero automático de un banco. Se nos permite introducir nuestro código hasta tres veces. Si al tercer intento no introducimos el código correctamente, no se nos permitirá entrar al sistema. Supongamos que nuestro código es 6767. Podríamos implementarlo utilizando un contador (variable **numIntentos**) que llegara hasta tres y a partir de ahí se saliera del bucle:

```
// Declaración de variables
Scanner teclado = new Scanner(System.in);
final int CODIGO= 6767; // Constante que contiene el código correcto
int codigoIntroducido;  // Código introducio por el usuario
int numIntentos=0;      // Contador que representa el número de intentos

// Entrada de datos
do {
    System.out.print ("Introduzca código (entre 0 y 9999): ");
    codigoIntroducido= teclado.nextInt();
    numIntentos++;
    if (codigoIntroducido != CODIGO)
        System.out.println ("Código incorrecto.");
} while (codigoIntroducido != CODIGO && numIntentos<3); // Mientras el código sea incorrecto y no
hayamos llegado al límite de intentos

// Comprobación de código correcto
if (codigoIntroducido == CODIGO)
    System.out.println ("Código correcto. Acceso concedido.");
else
    System.out.println ("Número de intentos superado. Acceso bloqueado");
```

De esta manera, sabemos que el bucle se puede ejecutar entre una vez como mínimo (por ser **do-while**) y tres veces como máximo (que es el número máximo de intentos permitidos). Eso lo controlamos mediante la condición **(codigoIntroducido != CODIGO && numIntentos<3)**, que mientras sea **true** hará que se vuelva a ejecutar el cuerpo del bucle. Si una vez salgamos del bucle

Puedes ver que dentro de una estructura iterativa (bucle) puede haber estructuras de tipo condicional (**if**, **if-else**, **switch**) sin problema.

También podríamos haber implementado este programa utilizando una variable de tipo **boolean** para evitar realizar varias veces comprobaciones similares:

```java
// Declaración de variables
Scanner teclado = new Scanner(System.in);
final int CODIGO= 6767; // Constante que contiene el código correcto
int codigoIntroducido;  // Código introducio por el usuario
int numIntentos=0;      // Contador que representa el número de intentos
boolean codigoCorrecto; // Indica si se ha llegado a introducir el código correcto

// Entrada de datos
codigoCorrecto= false;  // Comenzamos asumiendo que no tenemos un código correcto
do {
    System.out.print ("Introduzca código (entre 0 y 9999): ");
    codigoIntroducido= teclado.nextInt();
    numIntentos++;
    if (codigoIntroducido == CODIGO) {
        codigoCorrecto= true;  // Si el código es correcto, esta variable pasa a true
        System.out.println ("Código incorrecto.");
    }
} while (!codigoCorrecto && numIntentos<3); // Mientras el código sea incorrecto y no hayamos
llegado al límite de intentos

// Comprobación de código correcto
if (codigoCorrecto)
    System.out.println ("Número de intentos superado. Acceso bloqueado");
else
    System.out.println ("Código correcto. Acceso concedido.");
```

Ejercicio Resuelto

Escribe un programa que solicite dos números por teclado (inicio y fin, donde inicio debería ser menor o igual que fin) y muestre por pantalla cuántos múltiplos de tres hay entre esos dos números, ambos incluidos. Utiliza una variable de tipo **contador** para ir contándolos.

Aquí tienes un ejemplo de una posible ejecución del programa:

```
Introduzca el inicio: 2
Introduzca el fin: 15


Entre 2 y 15 hay 5 múltiplos de tres.
```

Vamos a utilizar dos contadores, uno para ir desde el inicio hasta el fin (tal y como se ha hecho en otros ejercicios) y otro para ir contando o "marcando" cuántos múltiplos de tres vamos encontrando:

- variable **contador**;
- variable **contadorMultiplos3**.

```java
// Declaración de variables
// ------------------------
Scanner teclado = new Scanner(System.in);
int inicio, fin;  // Entradas
int contador, contadorMultiplos3; // Dos contadores


// Entrada de datos
// ----------------
System.out.print ("Introduzca el inicio: ");
inicio = teclado.nextInt();
System.out.print("Introduzca el fin: ");
fin = teclado.nextInt();
```

```
// Procesamiento
// -------------
// Iniciamos contadores
contador= inicio;
contadorMultiplos3=0;
// Recorremos números desde inicio hasta fin
while (contador <= fin) {
    if (contador % 3 == 0)  // Si alguno es múltiplo de 3, incrementamos el contador de múltiplos
        contadorMultiplos3++;
    contador++;  // Incrementamos el contador que va desde inicio hasta fin
}
// Salida de resultados
// --------------------
System.out.println ("\nEntre " + inicio + " y " + fin + " hay " + contadorMultiplos3 + " múltiplos de tres.");
System.out.println ();
```

Ejercicio resuelto

Escribe un programa que muestre por pantalla una cuenta atrás que vaya de diez en diez, comenzando en 100 y terminando en 0. La salida debería ser algo similar a lo siguiente:

```
Cuenta atrás desde 100 hasta 0, de 10 en 10.
100 90 80 70 60 50 40 30 20 10 0
```

Un posible programa que resuelva el problema prodía ser el siguiente:

```
// Declaración de variables
int contador;
// No hay entrada de datos: tenemos toda la información que necesitamos
// Procesamiento y salida de resultados a la vez: cuenta atrás
System.out.println ("Cuenta atrás desde 100 hasta 0, de 10 en 10.");
contador= 100; // Iniciamos el contador a 100
while (contador >=0 ) { // Mientras el contador se mayor o igual que cero seguimos iterando el bucle
    System.out.print (contador + " ");
    contador -= 10; // Vamos restando 10 en cada iteración
}
System.out.println ();
```

REFLEXIONA:

Un contador no siempre tiene por qué incrementarse. Podemos encontrarnos con casos en los que el contador comience con un determinado valor y vaya decrementándose hasta otro valor (podríamos decir que se trata de un "descontador", pues va "descontando" en lugar de "contando", o bien "contando hacia atrás").

Por otro lado, **un contador no siempre tiene por qué ir de uno en uno**. Puede ir de dos en dos, tres en tres, o en general al ritmo que se considere oportuno.

4.4 CONCEPTO DE ACUMULADOR.

Del mismo modo que hemos hablado de contadores, que son variables que se van incrementando (o decrementando) según un determinado ritmo o bien cuando se produce alguna circunstancia (se está "contando" algo), también podemos encontrarnos con la necesidad de ir acumulando en una variable los resultados que vayamos obteniendo a lo largo de la ejecución de un bucle. Es decir, que en lugar de ir sustituyendo el valor anterior por un valor nuevo, lo "acumulamos" o "sumamos" en la variable de algún modo. A este tipo de variables se les suele conocer como **acumuladores**.

Los acumuladores más habituales son los **aditivos** o **sumativos**. Por ejemplo, imagina que queremos calcular la suma de todos los números que hay entre un número de inicio y un número de fin, ambos incluidos. Por ejemplo, la suma entre los números 1 y 5 sería 1+2+3+4+5 = 15. ¿Cómo podríamos automatizar este proceso mediante un programa en Java? Para lograr algo así necesitaríamos un contador que fuera desde 1 hasta 5 y un acumulador que fuera incorporando a lo que ya tienes cada uno de los nuevos valores que va adquiriendo el contador.

Podemos verlo paso a paso:

1. Iniciamos el contador (variable **contador**) y el acumulador (variable **suma**):

```
int contador=1; // Contador que irá desde inicio (1) hasta fin (5)
int suma=0;     // Acumulador que irá sumando de manera consecutiva los distintos valores que vaya tomando del contador
```

2. Recorremos todos los números que haya entre 1 y 5 (inicio y fin) con un bucle while

```
while (contador <= 5)
```

3. En el cuerpo del bucle vamos realizando la suma acumulada e incrementado el contador:

```
{
    suma += contador;
    contador++;
}
```

4. Mostramos por pantalla el resultado final:

```
System.out.println ("La suma de los números entre 1 y 5 es " + suma);
```

Uniéndolo todo tendríamos:

```
// Declaración de variables
int contador; // Contador que irá desde inicio (1) hasta fin (5)
int suma;     // Acumulador que irá sumando de manera consecutiva los distintos valores que vaya tomando del contador

// Iniciamos contadores
contador=1; // Iniciamos el contador a 1
suma=0;     // Iniciamos el acumulador a 0, para ir sumando todo lo que se vaya recorriendo

// Realizamos el recorrido
while (contador <= 5) {  // Mientras contador no supere el valor 5...
    suma += contador; // En cada iteración: cumulamos en suma el valor de cada contador
    contador++;       // En cada iteración: incrementamos en 1 el contador
}

// Mostramos el resultado por pantalla
System.out.println ("La suma de los números entre 1 y 5 es " + suma);
```

Ejercicio Resuelto

Escribe un programa que solicite dos números por teclado (inicio y fin, donde inicio debería ser menor o igual que fin) y muestre por pantalla la suma de los múltiplos de tres hay entre esos dos números, ambos incluidos. Utiliza una variable de tipo **acumulador** para calcular esa suma.

Aquí tienes un ejemplo de una posible ejecución del programa:

```
Introduzca el inicio: 2
Introduzca el fin: 15


La suma de los múltiplos de 3 entre 2 y 15 es 45.
```

Vamos a utilizar un contador para ir desde el inicio hasta el fin (tal y como se ha hecho en otros ejercicios) y un acumulador para ir sumando o "acumulando" cuántos múltiplos de tres vamos encontrando:

- variable **contador**;
- variable **sumaMultiplos3**.

```
// Declaración de variables
// ------------------------
Scanner teclado = new Scanner(System.in);
int inicio, fin;    // Entradas
int contador;       // contador
int sumaMultiplos3; // acumulador
// Entrada de datos
// ----------------
System.out.print ("Introduzca el inicio: ");
inicio = teclado.nextInt();
System.out.print("Introduzca el fin: ");
fin = teclado.nextInt();
// Procesamiento
// -------------
contador= inicio;  // Iniciamos contador
sumaMultiplos3=0;  // Iniciamos acumulador
// Recorremos números desde inicio hasta fin
while (contador <= fin) {
    if (contador % 3 == 0)  // Si alguno es múltiplo de 3, lo sumamos al acumulador (lo
"acumulamos")
        sumaMultiplos3 += contador;
    contador++;  // Incrementamos el contador que va desde inicio hasta fin
}
// Salida de resultados
// --------------------
System.out.println ("La suma de los múltiplos de 3 entre " + inicio + " y " + fin + " es " +
sumaMultiplos3 + ".");
System.out.println ();
```

REFLEXIONA:

La forma de acumular no tiene por qué ser siempre sumando (acumulación aditiva o sumativa). Podría ser también, por ejemplo, multiplicando (acumulación multiplicativa). En tal caso es muy importante que el valor inicial del acumulador no sea cero, sino uno, pues si multiplicas por cero el resultado siempre será cero. En general, el valor inicial de un acumulador debe ser el elemento neutro del operador que se vaya a utilizar para "acumular" (0 para la suma o la resta, 1 para el producto o la división, la cadena vacía para la concatenación, etc.).

Ejercicio Resuelto

El factorial de un número natural n se calcula multiplicando todos los números que van desde 1 hasta n y se representa por el símbolo de la exclamación (n!).

Por ejemplo 4!= 1×2×3×4 = 24.

Escribe un programa en Java que solicite un número natural positivo n y calcule y muestre por pantalla el valor de su factorial n!

Aquí tienes un ejemplo de una posible ejecución:

```
Introduzca un número n para calcular su factorial: 6

El factorial n! es 720.
```

Un posible programa que resuelva el problema prodía ser el siguiente:

```java
// Declaración de variables
// ------------------------
Scanner teclado = new Scanner(System.in);
int numero;            // Entrada
int contador;          // contador
int productoAcumulado; // acumulador

// Entrada de datos
// ----------------
System.out.print ("Introduzca un número n para calcular su factorial: ");
numero = teclado.nextInt();

// Procesamiento
// -------------

contador= 1;          // Iniciamos contador

productoAcumulado=1;  // Iniciamos acumulador (no puede iniciarse a cero porque es multiplicativo)

// Recorremos números desde inicio hasta fin
while (contador <= numero) {
    productoAcumulado *= contador;  // Vamos acumulando productos
    contador++;  // Incrementamos el contador que va desde inicio hasta fin
}

// Salida de resultados
// --------------------
System.out.println ("El factorial n! es " + productoAcumulado + ".");
System.out.println ();
```

Amplía el ejercicio anterior para que además de calcular el factorial de un número, también genere una cadena de caracteres a base de concatenaciones que represente los cálculos que se han tenido que realizar para obtenerlo. Por ejemplo, si para obtener el factorial de 4 (4!) hay que llevar a cabo las operaciones 1×2×3×4, habría que generar una cadena con el contenido **"1*2*3*4"**. Esta cadena la podemos ir creando a base de concatenaciones a la vez que se van realizando los cálculos. Sería un ejemplo de acumulador por yuxtaposición o concatenación.

Aquí tienes un ejemplo de una posible ejecución:

```
Introduzca el número n para calcular su factorial: 6

El factorial n! es 1*2*3*4*5*6 = 720.
```

Para generar la cadena resultado que se nos pide deberíamos comenzar con una cadena (**String**) vacía a la que le vamos concatenando ("acumulando") cada uno de los números que se van multiplicando junto con el símbolo asterisco de multiplicación (**"*"**).

```
String procesoCalculo; // acumulador para ir guardando los pasos del proceso

procesoCalculo= "";   // Iniciamos acumulador a la cadena vacía (para poder empezar a concatenar a algo)
```

Y dentro del bucle de cálculo, habrá que añadir la concatenación de cada número que se va multiplicando:

```
procesoCalculo += contador + "*";  // Vamos acumulando cada paso (multiplicación) realizado
```

Dentro del contexto de bucle quedaría:

```
while (contador <= numero) {
    productoAcumulado *= contador;     // Vamos acumulando productos
    procesoCalculo += contador + "*";  // Vamos acumulando cada paso (multiplicación) realizado
    contador++;  // Incrementamos el contador que va desde inicio hasta fin
}
```

El inconveniente que tiene hacer esto así es que generaríamos un asterisco de más en el último paso. Por ejemplo, si calculamos el factorial de 4, generaríamos la cadena **"1*2*3*4*"**. Nos sobra el último asterisco. Eso podemos intentar evitarlo incluyendo un **if** dentro del bucle para que solo si **contador** está por debajo del **numero** se concatene el asterisco. De este modo en el último paso no se concatena un asterisco, tan solo el valor de **contador**.

```
while (contador <= numero) {
    productoAcumulado *= contador;     // Vamos acumulando productos
    procesoCalculo += contador;  // Vamos acumulando cada paso (multiplicación) realizado
    if (contador < numero)
        procesoCalculo += "*"; // Concatenamos asterisco salvo para el último factor
    contador++;  // Incrementamos el contador que va desde inicio hasta fin
}
```

El programa completo podría quedar entonces así:

```
// Declaración de variables
// ------------------------
Scanner teclado = new Scanner(System.in);
int numero;            // Entrada
int contador;          // contador
int productoAcumulado; // acumulador para obtener el resultado final
String procesoCalculo; // acumulador para ir guardando los pasos del proceso

// Entrada de datos
// ----------------
System.out.print ("Introduzca el número n para calcular su factorial: ");
numero = teclado.nextInt();
```

```
// Procesamiento
// -------------
contador= 1;          // Iniciamos contador
productoAcumulado=1;  // Iniciamos acumulador (no puede iniciarse a cero porque es multiplicativo)
procesoCalculo= "";   // Iniciamos acumulador a la cadena vacía (para poder empezar a concatenar a
algo)
// Recorremos números desde inicio hasta fin
while (contador <= numero) {
    productoAcumulado *= contador;     // Vamos acumulando productos
    procesoCalculo += contador;  // Vamos acumulando cada paso (multiplicación) realizado
    if (contador < numero)
        procesoCalculo += "*"; // Concatenamos asterisco salvo para el último factor
    contador++;  // Incrementamos el contador que va desde inicio hasta fin
}

// Salida de resultados
// --------------------
System.out.println ("El factorial n! es " + procesoCalculo + " = " + productoAcumulado + ".");
System.out.println ();
```

Ejercicio Resuelto

Una manera de calcular el número de cifras que tiene un número natural es ir realizando la división entera entre diez hasta que el cociente sea cero. El número de veces que hayamos podido dividir será el número de cifras que tiene el número.
Por ejemplo, si tenemos el número 3521 y vamos dividiendo sucesivamente entre diez hasta que obtengamos cero tendríamos:
3521 / 10 = 352 → 352/10 = 35 → 35/10 = 3 → 3/10=0
Dado que hemos podido dividir cuatro veces, sabemos que el número tiene cuatro cifras.
Escribe un programa en Java que solicite un número natural positivo n y calcule y muestre por pantalla su número de cifras aplicando el anterior proceso de cálculo. Recuerda que debes trabajar con números enteros y no reales.
Aquí tienes un ejemplo de una posible ejecución:

```
Introduzca el número n para calcular su número de cifras: 3521

El número de cifras de n es 4.
```

En este caso tendremos un contador para el número de cifras y un acumulador (o "desacumulador" si prefieres verlo así, pues cada vez se irá reduciendo) para lo que vaya quedando del número original tras los sucesivos cocientes.

```
int numCifras; // contador (para ir "contando" las cifras)

int residuo;   // Acumulador (o "desacumulador") que almacena lo que va quedando del número tras
cada división
```

La inicialización de estos elementos será:

```
numCifras= 0;      // Iniciamos contador (al final contendrá el número de cifras)

residuo= numero;   // Iniciamos acumulador (al valor inicial del número)
```

Y el proceso de cálculo consistirá en la sucesiva división entre 10 hasta que el cociente sea cero. El número de veces que se haya podido dividir (contado por el contador numCifras) será el número de cifras del número:

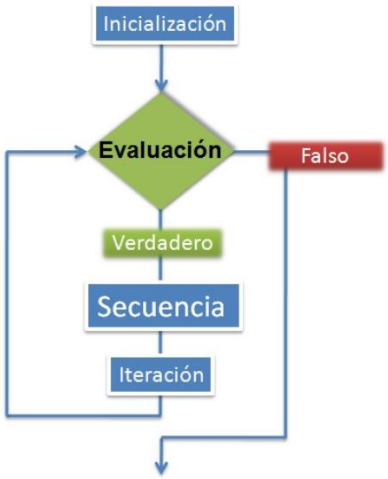
```
while (residuo > 0) {

    residuo = residuo / 10;     // Vamos diviendo entre 10

    numCifras++;  // Incrementamos el contador por cada vez que se pueda dividir

}
```

El programa completo podría quedar así:

```
// Declaración de variables
// ------------------------
Scanner teclado = new Scanner(System.in);
int numero;     // Entrada
int numCifras;  // contador (para ir "contando" las cifras)
int residuo;    // Acumulador (o "desacumulador") para almacenar lo que va quedando del número
tras cada división

// Entrada de datos
// ----------------
System.out.print ("Introduzca el número n para calcular su número de cifras: ");
numero = teclado.nextInt();

// Procesamiento
// -------------
numCifras= 0;      // Iniciamos contador (al final contendrá el número de cifras)
residuo= numero;   // Iniciamos acumulador (al valor inicial del número)
// Vamos dividiendo mientras el residuo sea mayor que cero
while (residuo > 0) {
    residuo = residuo / 10;     // Vamos diviendo entre 10
    numCifras++;  // Incrementamos el contador por cada vez que se pueda dividir
}

// Salida de resultados
// --------------------
System.out.println ("El número de cifras de n es " + numCifras + ".");
System.out.println ();
```

4.5 ESTRUCTURA REPETITIVA FOR.

Hemos indicado anteriormente que el bucle **for** es un bucle **controlado por contador**. ¿Recuerdas lo que significaba un contador?



Este tipo de bucle tiene las siguientes características:

- Se ejecuta **un número determinado de veces conocido** a priori.
- Utiliza un **contador** (una variable usada como contador o índice) que controla las iteraciones que se van haciendo del bucle.

En general, existen **tres operaciones** que se llevan a cabo sobre la variable contador que controla la ejecución en este tipo de bucles:

- Se **inicializa** la variable contador.
- Se **evalúa** el valor de la variable contador, por medio de una comparación de su valor con el número de iteraciones especificado para saber si hay que continuar con otra nueva iteración.
- Se **actualiza** con incrementos o decrementos el valor del contador, en cada una de las iteraciones.

En realidad esas tres operaciones también las has realizado cuando has usado contadores en bucles de tipo **while** o **do-while**. La diferencia en este tipo de bucles es que sistematizamos su utilización.

Aspectos importantes:

- ✓ La **inicialización del contador** debe realizase correctamente para hacer posible que el bucle se ejecute al menos la primera repetición de su código interno, aunque puede haber casos en los que no queramos ejecutarlo ninguna vez si la condición es de partida falsa.
- ✓ La **condición de terminación** del bucle es importante establecerla cuidadosamente, ya que si no, podemos caer en la creación de un bucle infinito, cuestión que se debe evitar por todos los medios.
- ✓ Es necesario estudiar el **número de veces que se repite el bucle**, pues debe ajustarse al número de veces estipulado.

Sintáxis	Funcionamiento

Estructura for con una única sentencia (no necesita las llaves):
`for (inicialización ; condición ; incremento)`
`  sentencia;`
Estructura for con un bloque de sentencias:
`for (inicialización ; condición ; incremento){`
`  sentencia_1;`
`  sentencia_2;`
`  ...`
`  sentencia_N;`
`}`
	✓ Donde **inicialización** es una expresión en la que se inicializa una variable de control, que será la encargada de controlar el final del bucle.
	✓ Donde **condición** es una expresión que evaluará la variable de control. Mientras la condición sea verdadera (condición de "continuidad"), el cuerpo del bucle estará repitiéndose. Cuando la condición deje de cumplirse, terminará la ejecución del bucle.
	✓ Donde **incremento** indica la manera en la que la variable de control va cambiando en cada iteración del bucle. Podrá ser en realidad, mediante incremento o decremento, y no solo de uno en uno.

En este caso, la sintaxis es algo diferente a la que hemos visto en las estructuras de tipo **while** y **do-while**. Veamos el mismo ejemplo del contador de 1 a 5 utilizando un bucle **for**.

Aquí tanto la inicialización (**numero=1**) como la condición (**numero<=5**) y el incremento (**numero++**) los tenemos en la propia cabecera de la estructura **for**:

```
for ( numero=1 ; numero<=5 ; numero++ )
```

Y en el cuerpo tendríamos únicamente la parte de mostrar por pantalla el valor de la variable contador (**numero**):

```
{
    System.out.println (numero);
}
```

Si lo unimos todo:

```
int numero;
for (numero=1; numero<=5; numero++) {
    System.out.println (numero);
}
```

Si así lo consideras, podrías incluso declarar el contador dentro de la propia estructura **for**. Ahora bien, esa variable solo existirá dentro del bloque **for** y no se podrá acceder a ella desde fuera. Por ejemplo:

```
// Aquí aún no existe la variable numero
for (int numero=1; numero<=5; numero++) { // Aquí se declara la variable número
    System.out.println (numero);
}
// Aquí deja de existir la variable numero
```

De hecho, si intentas utilizar la variable **numero** en la línea 1 o la línea 5, se produciría un error de compilación y no se podría ejecutar el programa, pues esa variable no existe fuera del contexto del ese bucle.

Por último, recuerda que si el bloque es de una única línea, puedes omitir las llaves:

```
// Aquí aún no existe la variable numero
for (int numero=1; numero<=5; numero++) // Aquí se declara la variable numero
    System.out.println (numero);
// Aquí deja de existir la variable numero
```

Ejercicio Resuelto

Escribe un programa que solicite dos números por teclado (inicio y fin) y muestre por pantalla todos los números que van desde inicio hasta fin, todos en una misma línea. Este ejercicio ya lo hemos planteado usando bucles **while** y **do-while**. En este caso debes utilizar un bucle **for**.

Aquí tienes un ejemplo de una posible ejecución del programa:

```
Introduzca el inicio: 4
Introduzca el fin: 11


Secuencia de números desde 4 hasta 11
4 5 6 7 8 9 10 11
```

Aquí tienes una muestra de cómo podría quedar tu programa:

```
// Declaración de variables
Scanner teclado = new Scanner(System.in);
int inicio, fin, numero;

// Entrada de datos
System.out.print ("Introduzca el inicio: ");
inicio = teclado.nextInt();

System.out.print("Introduzca el fin: ");
fin = teclado.nextInt();

// Procesamiento y salida de resultados
System.out.println ("\nSecuencia de números desde " + inicio + " hasta " + fin);
for (numero=inicio; numero <= fin; numero++) {
    System.out.print (numero + " ");
}
System.out.println ();
```

En las estructuras de tipo **for** en Java podemos prescindir de alguno de los tres elementos que la forman e incluso, podemos utilizar más de una variable contadora separando estas por comas. Se trata de algo que no usarás a menudo, pero que es bueno que al menos veas una vez por si te lo encuentras en código ya escrito.

Veamos algunos ejemplos:

1. Inicialización de más de una variable:

```
for (contador1=0, contador2=0 ; contador1 <= 10 ; contador1++)
```

2. Condiciones complejas donde se usan operadores lógicos:

```
for (contador1=0 ; contador1 <= 10 && limite<=10; contador1++)
```

3. Incremento o decremento de más de un contador:

```
for (contador1=0, contador2=0; contador1 <= 10 && contador2<=10; contador1++, contador2++)
```

4. Ausencia de inicialización:

```
for ( ; contador1 <= 10 ; contador1++)
```

5. Ausencia de inicializaciones y de incrementos o decrementos:

```
for ( ; contador1 <= 10 ; )
```

Y muchos otros ejemplos que se te puedan ocurrir. Lo que sí debes de tener en cuenta es que si omites alguno de sus componentes esa función deberá realizarse en otra parte del código. Por ejemplo, si omites la inicialización, la variable contador deberás inicializarla tú antes al valor adecuado para que el bucle funcione como tú quieres que lo haga. En otros casos, si omites el incremento/decremento, será tu responsabilidad en el cuerpo del bucle hacer que el contador pueda modificar su valor en algún momento para que el ciclo no sea infinito.

Ejercicio Resuelto

Al igual que se pedía en apartados anteriores, ahora vamos os vamos a pedir que realicéis el ejercicio de la tabla de multiplicar del número 7, pero usando un bucle una estructura repetitiva tipo **for**.

Una posible solución del ejercicio podría ser la siguiente:

```
/**
 * Mostrar la tabla de 7 usando una estructura repetitiva for.
 * @author Profesor
 */
public class RepetitivaFor {

    /**
     * En esta solución se utiliza la estructura repetitiva for para representar
     * en pantalla la tabla de multiplicar del siete.
     */
    public static void main(String[] args) {
        // Declaración e inicialización de variables
        int numero = 7;
        int contador;
        int resultado = 0;

        /* Paso 1. Mostrar la cabecera de la tabla */
        System.out.println("Tabla de multiplicar del " + numero);
        System.out.println("............................ ");

        /* Paso 2. Calcular la tabla de multiplicar del 7 usando un bucle for.
         * La cabecera del bucle incorpora la inicialización de la variable
         * de control, la condición de multiplicación hasta el 10 y el
         * incremento de dicha variable de uno en uno en cada iteración del bucle.
         * En este caso contador++ incrementará en una unidad el valor de
         * dicha variable.
         */
        for (contador = 1; contador <= 10; contador++) {
            resultado = contador * numero;
            System.out.println(numero + " x " + contador + " = " + resultado);
        }

        /* Nota: A través del operador + aplicado a cadenas de caracteres,
         * concatenamos los valores de las variables con las cadenas de
         * caracteres que necesitamos para representar correctamente la
         * salida de cada multiplicación.
         */
    }
}
```

Ejercicio Propuesto

Escribe en Java un programa que solicite un **número n** para calcular la **tabla de multiplicar de ese número n** usando un bucle tipo **for**.

Una vez lo tengas hecho, añade a la entrada de datos una comprobación para que el número n introducido esté obligatoriamente entre 1 y 10. Si no es así, se volverá a solicitar el número hasta que esté dentro de ese rango.

Recomendación

Error común de programación

- ✓ Utilizar un operador relacional incorrecto o un valor final incorrecto de un contador de ciclo en la condición de continuación de ciclo de una instrucción de repetición puede producir un error por desplazamiento en 1.
- ✓ Utilizar comas en vez de los dos signos de punto y coma requeridos en el encabezado de una instrucción **for** es un error de sintaxis.
- ✓ Cuando se declara la variable de control de una instrucción **for** en la sección de inicialización del encabezado del **for**, si se utiliza la variable de control fuera del cuerpo **for** se produce un error de compilación.
- ✓ Colocar un punto y coma inmediatamente a la derecha del paréntesis derecho del encabezado de un **for** convierte el cuerpo de ese **for** en una instrucción vacía. Por lo general, se trata de un error lógico.
- ✓ No utilizar el operador relacional apropiado en la condición de continuación de un ciclo que cuente en forma regresiva (como usar **i <= 1** en lugar de **i >= 1** en un ciclo que cuente en forma regresiva hasta llegar a 1) es generalmente un error lógico.

Buena práctica de programación

- ✓ Utilizar el valor final en la condición de una instrucción de una instrucción **for** (o **while**) con el operador relacional <= nos ayuda a evitar errores por desplazamiento en 1. Por ejemplo, para un ciclo que imprime valores del 1 al 10, la condición de continuación del ciclo debe ser contador <= 10, en vez de contador < 10 (lo cual produce un error por desplazamiento en uno) o contador < 11 (que es correcto). Muchos programadores prefieren el llamado conteo con base 0, en el cual para contar 10 veces, contador se inicializaría a cero y la prueba de continuación del ciclo sería contador < 10.
- ✓ Limita el tamaño de los encabezados de las instrucciones de control a una sola línea, si es posible.

Notas para prevenir errores

- ✓ Los ciclos infinitos ocurren cuando la condición de continuación del ciclo en una instrucción de repetición nunca se vuelve false. Para evitar esta situación en un ciclo controlado por un contador, debes asegurarte que la variable de control se incremente (o decremente) durante cada iteración del ciclo.

Ejercicios Resueltos

Vamos a intentar resolver algunos de los ejercicios planteados en apartados anteriores utilizando la estructura de tipo **for**.

1. Escribe un programa en Java que muestre por pantalla una cuenta atrás que vaya de diez en diez, comenzando en 100 y terminando en 0.

La salida debería ser algo similar a lo siguiente:

```
Cuenta atrás desde 100 hasta 0, de 10 en 10.
100 90 80 70 60 50 40 30 20 10 0
```

```
System.out.println ("Cuenta atrás desde 100 hasta 0, de 10 en 10.");
for (int contador=100; contador>=0; contador -=10 ) {
    System.out.print (contador + " ");
}
System.out.println ();
```

2. Escribe un programa en Java que solicite un número natural positivo n y calcule y muestre por pantalla el valor de su factorial n!

Aquí tienes un ejemplo de una posible ejecución:

```
Introduzca un número n para calcular su factorial: 6
El factorial n! es 720.
```

```
// Declaración de variables
// ------------------------
Scanner teclado = new Scanner(System.in);
int numero;            // Entrada
int productoAcumulado; // acumulador

// Entrada de datos
// ----------------
System.out.print ("Introduzca un número n para calcular su factorial: ");
numero = teclado.nextInt();


// Procesamiento
// -------------
          // Iniciamos contador
productoAcumulado=1;  // Iniciamos acumulador (no puede iniciarse a cero porque es multiplicativo)
// Recorremos números desde inicio hasta fin
for (int contador= 1 ; contador <= numero; contador++) {
    productoAcumulado *= contador;  // Vamos acumulando productos
}

// Salida de resultados
// --------------------
System.out.println ("El factorial n! es " + productoAcumulado + ".");
System.out.println ();
```

3. Escribe un programa en Java que solicite un número natural positivo n y calcule y muestre por pantalla su número de cifras aplicando el anterior proceso de cálculo. Recuerda que debes trabajar con números enteros y no reales.

Aquí tienes un ejemplo de una posible ejecución:

```
Introduzca el número n para calcular su número de cifras: 29000
El número de cifras de n es 5.
```

```java
// Declaración de variables
// ------------------------
Scanner teclado = new Scanner(System.in);
int numero;     // Entrada
int numCifras;  // contador (para ir "contando" las cifras)

// Entrada de datos
// ----------------
System.out.print ("Introduzca el número n para calcular su número de cifras: ");
numero = teclado.nextInt();


// Procesamiento
// -------------
numCifras= 0;
// Vamos dividiendo mientras el residuo sea mayor que cero
for (int residuo=numero; residuo > 0 ; residuo /= 10) {
    numCifras++;  // Incrementamos el contador por cada vez que se pueda dividir
}

// Salida de resultados
// --------------------
System.out.println ("El número de cifras de n es " + numCifras + ".");
System.out.println ();
```

Para saber más.

Junto a la estructura **for**, tenemos la estructura **for/in** o **foeach**, que también se considera un bucle controlado por contador. Este bucle es una mejora incorporada desde la versión 5.0. de Java, por lo que no funcionará en versiones más antiguas del lenguaje.

Este tipo de bucles permite realizar recorridos **sobre arrays y colecciones de objetos**. Los **arrays** son colecciones de variables que tienen el mismo tipo y se referencian por un nombre común junto a un índice que indica el lugar que ocupa el elemento dentro del **array**. Veremos este tipo de bucles más especializados a partir de la unidad 4.

5. ESTRUCTURAS DE SALTO INCONDICIONAL.

En la gran mayoría de libros de programación y publicaciones de Internet, siempre se nos recomienda que **prescindamos de sentencias de salto incondicional**, es más, **se desaconseja su uso por provocar una mala estructuración del código y un incremento en la dificultad para el mantenimiento** del mismo.

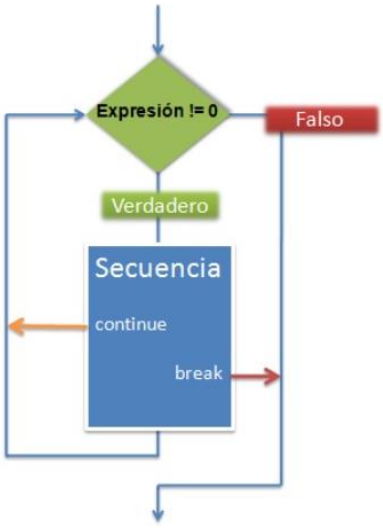
Pero Java incorpora ciertas sentencias o estructuras de salto que es necesario conocer, que en algunos casos son imprescindibles, y que por tanto son útiles en algunas partes de nuestros programas.

En Java, las estructuras de salto incondicional están representadas por las sentencias **break**, **continue**, las **etiquetas de salto** y la sentencia **return**. Esta última sentencia la estudiaremos más adelante cuando aprendamos a implementar métodos en la unidad 5.

No obstante, esos usos deben ser siempre compatibles con los **principios de la programación estructurada,** que promueven seguir una serie de reglas:

- ✓ **Limitar el uso de estructuras de control** a las tres estudidas hasta ahora: **secuencial, selectiva y repetitiva**.
- ✓ **Mantener el principio de "una entrada - una salida"**. Eso implica que cualquier bloque de código debe tener una única entrada y una única salida. Esto desde luego, se consigue limitando el uso de las tres estructuras anteriores, ya que todas tienen una entrada y una salida, pero también hay que tenerlo presente en los casos en los que hagamos uso de sentencias de salto. Un buen ejemplo sería la sentencia **switch**, que requiere usar los **break** para evita rel efecto de "ejecución en cascada", pero que no por ello rompe el principio de la entrada única y salida única.
- ✓ **Evitar saltos a regiones remotas de código.** Incluso en los casos en los que se estime que usar sentencias de salto incondicional puede mejorar la claridad del código, deben evitarse los saltos a regiones remotas del código, ya que resultan difíciles de seguir a la hora de hacer el mantenimiento del código y por tanto producen código poco claro, difícil de entender y de mantener, y costoso de desarrollar.

5.1 SENTENCIAS BREAK Y CONTINUE.



Se trata de dos instrucciones que permiten modificar el comportamiento de otras estructuras o sentencias de control, simplemente por el hecho de estar incluidas en algún punto de su secuencia de instrucciones.
La **sentencia break** incidirá sobre las estructuras de control **switch**, **while**, **for** y **do-while** del siguiente modo:

- ✓ Si aparece una sentencia **break** dentro de la secuencia de instrucciones de cualquiera de las estructuras mencionadas anteriormente, dicha estructura terminará inmediatamente.
- ✓ Si aparece una sentencia **break** dentro de un bucle anidado sólo finalizará ejecución del bucle más interno en el que se encuentra, el resto se ejecuta de forma normal.

Es decir, que **break** sirve para romper el flujo de control de un bucle, aunque no se haya cumplido la condición del bucle. Si colocamos un **break** dentro del código de un bucle, cuando se alcance el **break**, automáticamente se saldrá del bucle pasando a ejecutarse la siguiente instrucción inmediatamente después de él.
Aquí tienes un ejemplo de cómo se utilizaría la sentencia **break** dentro de un bucle **for**.

```
/**
 * Ejemplo de uso de la sentencia de salto break
 */
public class SentenciaBreak {
     public static void main(String[] args) {
        // Declaración de variables
            int contador;
            1
        //Procesamiento y salida de información
       /* Este bucle sólo se ejecutará en 6 ocasiones, ya que cuando
        * la variable contador sea igual a 7 encontraremos un break que
        * romperá el flujo del bucle, transfiriéndonos el control a la
        * sentencia que imprime el  mensaje  de Fin del programa.
        */
        for (contador=1;contador<=10;contador++){
            if (contador==7)
                 break;
            System.out.println ("Valor: " + contador); // Es una forma muy inapropiada de salir
del bucle!!
        }
        System.out.println ("Fin del programa");
     }
}
```

¡Recuerda!
Debemos saber cómo funciona **break**, pero su uso, salvo en el caso del **switch**, donde es obligado usarlo para evitar una "ejecución en cascada", se desaconseja, y siempre es evitable.
La salida de cualquier ciclo usando **break** es en general **una mala práctica** de programación, que esconde una mala planificación de la lógica asociada al ciclo. La **salida natural y única de cada ciclo** debe ser comprobando la condición de control del mismo, único punto donde debemos comprobar si ha llegado o no el momento de terminarlo. Cualquier comprobación de cualquier condición de salida dentro del cuerpo del bucle para forzar la salida del bucle desde el interior del mismo usando **break**, debería haberse incorporado a la condición de control del mismo, y supone ir abriendo puertas traseras de salida que hacen que el código se haga cada vez más complicado de entender y mantener. ¡¡Evita usar **break** siempre que sea posible!! Y salvo el caso de **switch**, siempre es posible.

La **sentencia continue** incidirá sobre las sentencias o estructuras de control **while**, **for** y **do-while** del siguiente modo:

- ✓ Si aparece una sentencia **continue** dentro de la secuencia de instrucciones de cualquiera de las sentencias anteriormente indicadas, dicha sentencia dará por terminada la iteración actual y se ejecuta una nueva iteración, evaluando de nuevo la expresión condicional del bucle.
- ✓ Si aparece en el interior de un bucle anidado solo detendrá la ejecución de la iteración del bucle más interno en el que se encuentra, el resto se ejecutaría de forma normal.

Es decir, la sentencia **continue** forzará a que se ejecute la siguiente iteración del bucle, ignorando y saltándose las instrucciones que pudiera haber después del **continue**, y hasta el final del código del bucle, para esta iteración.
Aquí tienes un ejemplo de cómo se utilizaría la sentencia **continue** dentro de un bucle **for** para mostrar por pantalla los números pares que hay entre el 1 y el 10:

```java
/**
 * Ejemplo de uso de la sentencia de salto continue
 */
public class EjemploSentenciaContinue {
     public static void main(String[] args) {
        // Declaración de variables
          int contador;
        //Procesamiento y salida de información
                    System.out.println ("Imprimiendo los números pares que hay del 1 al 10... ");
          for (contador=1;contador<=10;contador++){
            if (contador % 2 != 0)
                              continue;
            System.out.print(contador + " ");
          }
          System.out.println ("\nFin del programa");
        /* Las iteraciones del bucle que generarán la impresión de cada uno de los números
         * pares, serán aquellas en las que el resultado de calcular el resto de la división
         * entre 2 de cada valor de la variable contador, sea igual a 0.
         */
    }
}
```

¡¡Recuerda!!

Con la sentencia **continue**, también se desaconseja el uso. Igual que pasaba con el uso de **break**, es en general **una mala práctica** de programación, que esconde un mal diseño de la lógica asociada al ciclo. De hecho, la sentencia **continue**, como en el ejemplo de la imagen anterior, se pondría dentro de un **if** para que se ejecute dependiendo de una condición. La sentencia **continue** lo que hace implícitamente de hecho, es meter las demás sentencias que siguen a partir de ella en una "invisible" cláusula **else** del condicional, ya que sólo serán alcanzables y ejecutables en el caso de que la condición sea falsa y por tanto no se ejecute **continue**. Si un grupo de sentencias del bucle deben dejar de ejecutarse bajo ciertas circunstancias, lo que hay que hacer es incluirlas en un condicional que compruebe dicha condición, y que se salte esas sentencias cuando sea oportuno.

Reflexiona

¿Cómo reescribirías el código de los dos ejemplos de código anteriores para conseguir el mismo funcionamiento, pero sin usar sentencias break ni continue?

Prueba a cambiarlos y a ejecutarlos para comprobar que efectivamente el resultado es el esperado, y si tienes alguna duda, compártela con el resto de participantes del curso en el foro de la unidad.

Ejercicio Resuelto

Usando la sentencia **continue** dentro un bucle tipo **while**, intenta que se muestre la secuencia siguiente de 6 líneas:

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

Fíjate que en las líneas impares se muestra solo un asterísco, y en las pares, dos asteríscos. ¿Se te ocurre como resolver el problema usando un **continue**? Intenta primer solucionar el problema sin **continue** y luego con **continue**.

El siguiente código es una posible solución al problema **sin usar continue**:

```
public class SolucionEjercicioSinContinue {

    public static void main(String[] args)
    {
        int i=0;
        String cad="";
        while (i<6)
        {
            i++;
            if (i%2!=0) { System.out.println("*"); }
            else System.out.println("**");
        }
    }
}
```

El mismo ejercicio usando un **continue** se hace un poco más complejo de entender (es el tipo de soluciones que tenemos que evitar):

```
public class SolucionEjercicioContinue {

    public static void main(String[] args)
    {
        int i=0;
        String cad="";
        while (i<6)
        {
            i++;
            cad="*";
            if (i%2!=0) { System.out.println(cad); continue; }
            cad+="*";
            System.out.println(cad);
        }

    }
}
```

5.2 ETIQUETAS.

Ya lo indicábamos al comienzo del epígrafe dedicado a las estructuras de salto:

Los saltos incondicionales y, en especial, saltos a una etiqueta son totalmente desaconsejables.

Java permite asociar etiquetas cuando se va a realizar un salto, y por tanto es conveniente saber que existen y cómo se usan por si algún día te encuentras con un fragmento de código donde se utilice esta herramienta.

Las estructuras de salto **break** y **continue**, pueden tener asociadas etiquetas. Es a lo que se llama un **break etiquetado** o un **continue etiquetado**. Pero sólo podría estar indicado su uso cuando se hace necesario salir de bucles anidados hacia diferentes niveles, para indicar a qué nivel nos traslada una sentencia **break** o continue. No obstante, desde el momento que cualquier salida por la puerta trasera de un bucle usando **break** o **continue** es indeseable y evitable. Si las cosas se han hecho bien, no debería haber tales salidas y, por tanto, **no debería ser necesario recurrir a etiquetas.**

¿Y cómo se crea un salto a una etiqueta?

Crearemos la etiqueta mediante un **identificador seguido de dos puntos (:).** A continuación, se escriben las sentencias Java asociadas a dicha etiqueta encerradas entre llaves. Por así decirlo, la creación de una etiqueta es como fijar un punto de salto en el programa para poder saltar a él desde otro lugar de dicho programa.

¿Cómo se lleva a cabo el salto?

Es sencillo, en el lugar donde vayamos a colocar la sentencia **break** o **continue**, añadiremos detrás el identificador de la etiqueta. Con ello, conseguiremos que el salto se realice a un lugar determinado. La sintaxis será **break <etiqueta>**.

Quizá a quienes hayáis programado en HTML os suene esta herramienta, ya que tiene cierta similitud con las anclas que pueden crearse en el interior de una página web, a las que nos llevará el hiperenlace o link que hayamos asociado.

También para quienes hayáis creado alguna vez archivos por lotes o archivos batch bajo MS-DOS es probable que también os resulte familiar el uso de etiquetas, pues la sentencia **GOTO** que se utilizaba en este tipo de archivos, hacía saltar el flujo del programa al lugar donde se ubicaba la etiqueta que se indicara en dicha sentencia. A continuación, te ofrecemos un ejemplo de declaración y uso de etiquetas en un bucle. Como podrás apreciar, las sentencias asociadas a cada etiqueta están encerradas entre llaves para delimitar así su ámbito de acción.

```
/**
 * Ejemplo de uso de etiquetas en bucle
 */
public class EjemploUsoEtiquetas {
     public static void main(String[] args) {
              /*Creamos cabecera del bucle*/
         for (int i=1; i<3; i++){
             bloqueUno:{   //Creamos primera etiqueta
                 bloqueDos:{    //Creamos segunda etiqueta
                     System.out.println("Iteración: "+i);
                     if (i==1) break bloqueUno;   //Llevamos a cabo el primer salto
                     if (i==2) break bloqueDos;   //Llevamos a cabo el segundo salto
                 }
                 System.out.println("después del bloque dos");
             }
             System.out.println("después del bloque uno");
         }
         System.out.println("Fin del bucle  for");
     }
}
```

Ejercicio Resuelto

En este ejercicio que te proponemos ahora, te pedimos que diseñes un programa en Java que muestre por pantalla la siguiente secuencia de asteriscos:

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

Fíjate que en cada línea hay un asterisco más que en la anterior. Para realizar este ejercicio te proponemos que intentes realizarlo usando dos bucles **for**, uno anidado dentro de otro, y que apliques la concatenación de cadenas.

Lo ideal, dado que estas en el apartado del salto incondicional, es que intentes realizarlo primero usando una sentencia **break** y después sin usar un **break**.

Veamos como sería una posible solución al ejercicio usando una sentencia **break**:

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

        String cad="";
        for (int i=0;i<5;i++)

        {
            cad="";
            for (int j=0;j<5;j++) {

                cad=cad+"*";
                if (i==j) break;
            }
            System.out.println(cad);

        }
    }
}
```

La solución anterior parece ideal: cuando el contador **i** y el contador **j** son iguales, se realiza el **break** y se sale del bucle **for** interno; pero realmente hay una solución mucho mejor y más óptima sin usar la sentencia **break**:

```
public class SolucionEjercicioSinBreak {

    public static void main(String[] args)
    {
        String cad="";
        for (int i=0;i<5;i++)
        {
            cad="";
            for (int j=0;j<=i;j++) {
                cad=cad+"*";
            }
            System.out.println(cad);
        }
    }

}
```

En esta segunda solución, simplemente se ha modificado la condición de permanencia en el bucle **for** interno (**j<=i**), la cual es una solución elegante y fácil de entender.

6. PRUEBAS Y DEPURACIÓN DE CÓDIGO.

La **depuración de programas** es el proceso por el cual **se identifican y corrigen errores de programación**. Generalmente, en el argot de programación se utiliza la palabra debugging, que significa localización y eliminación de bichos (bugs) o errores de programa. A través de este proceso se descubren los errores y se identifica qué zonas del programa los producen. Hay tres etapas por las que un programa pasa cuando este es desarrollado y que pueden generar errores:

- ✓ **Compilación:** una vez que hemos terminado de escribir un programa, solemos pasar generalmente cierto tiempo eliminando errores de compilación para que el código pueda ejecutarse. Una vez que el programa es liberado de los errores de compilación, obtendremos de él algunos resultados visibles, pero quizá no haga aún lo que queremos.
- ✓ **Enlazado:** todos los programas hacen uso de librerías de métodos y otros utilizan métodos generados por los propios programadores. Un método es enlazado (linked) solo cuando este es llamado, durante el proceso de ejecución. Pero cuando el programa es compilado, se realizan comprobaciones para saber si los métodos llamados existen y sus parámetros son correctos en número y tipo. Así que, los errores de enlazado y de compilación son detectados antes de la fase de ejecución.
- ✓ **Ejecución:** cuando el programa entra en ejecución, es muy frecuente que este no funcione como se esperaba. De hecho, es normal que el programa falle. Algunos errores serán detectados automáticamente y el programador será informado, nos referimos a errores como acceder a una parte de un array que no existe (error de índices), por ejemplo. Otros son más sutiles y dan lugar simplemente a comportamientos no esperados, debido a la existencia de errores ocultos (bugs) en el programa. De ahí los términos bug y debugging. El problema de la depuración es que los síntomas de un posible error son generalmente poco claros, hay que recurrir a una labor de investigación para encontrar la causa.

Durante la **fase de compilación** estamos aún escribiendo código o bien ya lo hemos terminado de escribir, pero aún no hemos probado a ejecutar el programa. En esta fase los errores que podemos encontrar serán de tipo:

- ✓ **léxico**: uso de palabras o identificadores no reconocidos por el lenguaje Java, o que no forman parte de las bibliotecas utilizadas, ni tampoco son elementos definidos en el propio programa (por ejemplo nombres de variables). Algunos ejemplos de estos fallos son:
  - cuando te confundes al escribir una sentencia (por ejemplo **wile** en lugar de **while**);
  - cuando te equivocas al escribir el nombre de una variable (por ejemplo **cotador** en lugar de **contador**) o intentas usar una variable que no ha sido declarada en ese contexto o bloque;
  - cuando intentas llamar a algún elemento de biblioteca escribiendo su nombre erróneamente (por ejemplo si escribes **System.out.prtln()** en lugar de **System.out.println()**);
- ✓ **sintáctico**: estructuración errónea del código sin seguir las reglas del lenguaje. Algunos ejemplos podrían ser:
  - no cerrar todos los paréntesis que se han abierto en una expresión aritmética;
  - no cerrar todos los bloques que se han abierto (llaves) y en el orden apropiado;
  - no finalizar las sentencias con el carácter punto y coma;
- ✓ **semántico**: aplicación de operaciones que no tienen sentido en el lenguaje. Algunos de estos fallos podrían ser:
  - asignar a una variable entera un valor de tipo real. Se produciría una pérdida de precisión, así que el compilador nos obliga a realizar una conversión explícita (casting) para asegurarse de que es eso lo que realmente queremos hacer;
  - aplicar un operador a un tipo de dato que no corresponde (por ejemplo intentar aplicar el operador división "/" a un **boolean**).

Durante la fase de enlazado pueden producirse errores de integración a la hora de garantizar que todos los elementos que se usan en tu programa están disponibles. Por ejemplo si intentas usar la clase **Scanner** para leer del teclado y esa clase no está disponible, se produciría un error de enlazado.

Durante la fase de ejecución, una vez que el programa ha podido ser compilado y enlazado y está ya funcionando pueden producirse esencialmente dos tipos de errores:

- ✓ **Errores de propiamente de ejecución**, donde el sistema operativo (o en nuestro caso la máquina virtual de Java, que hace las veces de sistema operativo de nivel superior) "abortará" la ejecución y el programa se interrumpirá abruptamente. Una situación muy poco deseable que debemos evitar a toda costa. Estos errores los intentaremos prever mediante la gestión de excepciones, que ya veremos más adelante. Algunas circunstancias típicas que pueden producir errores de ejecución son la división por cero, el intentar ir más allá de los límites de una cadena o un array, intentar acceder a un archivo que no existe o sobre el que no se tiene permiso, etc.
- ✓ **Errores lógicos** de nuestro programa. Estos son los errores más difíciles de encontrar, pues el programa compila y se ejecuta perfectamente, pero no tiene el comportamiento que se esperaba de él. Eso suele ser debido a que nos hemos confundido en nuestro código, pero sigue siendo código correcto que no produce errores. Y dado que ni el compilador, ni la máquina virtual de Java, ni el sistema operativo son "adivinos", no saben que el programador se ha equivocado al implementar una determinada operación que está dando lugar a un resultado incorrecto.

Es para este último tipo de errores para los que la depuración es una herramienta esencial en el proceso de desarrollo, pues l**os errores de compilación y enlazado son sencillos de detectar**, ya que no llegamos a poder ejecutar el código y normalmente se nos indica con bastante precisión qué tipo de error se está produciendo. En el caso de los **errores de ejecución**, podemos intuir en qué parte del código se produce el error que hace que el programa aborte. Pero **en los errores lógicos ya no es tan sencillo detectar el posible fallo**, pues es posible que no siempre se produzca y que tan solo se dé bajo unas determinadas circunstancias. Además, aunque se produzca el fallo, el programa sigue funcionando sin problema, pues no es un error que necesariamente produzca un malfuncionamiento del equipo sobre el que se está ejecutando. Simplemente se están generando resultados incorrectos de vez en cuando. Eso hace que puedan tardarse días o incluso meses en detectar esos fallos, en algunos casos con la aplicación funcionando ya en producción.

La **depuración y la prueba de los programas** nos permitirán detectar y corregir este tipo de fallos. Suelen requerir una cantidad de tiempo considerable en comparación con el tiempo dedicado a la primera codificación del programa. Pero no te preocupes, es lo normal emplear más tiempo en este proceso.

La depuración de programas es algo así como ser doctor: existe un síntoma, hemos de encontrar la causa y entonces determinar el problema. Y, como ya se ha dicho, suele requerir una cantidad de tiempo considerable en comparación con el tiempo dedicado a la primera codificación del programa.

¿Y cómo llevamos a cabo la depuración de nuestros programas?

Pues a través del debugger o depurador del sistema de desarrollo Java que estemos utilizando. Este depurador será una herramienta que nos ayudará a eliminar posibles errores de nuestro programa. Podremos utilizar depuradores simples, como el **jdb propio de Java** basado en línea de órdenes (*command line*). O bien, utilizar el **depurador existente en nuestro IDE** (en nuestro caso NetBeans). Este último tipo de depuradores muestra los siguientes elementos en pantalla:

- ✓ El programa en funcionamiento.
- ✓ El código fuente del programa.
- ✓ Los nombres y valores actuales de las variables que se seleccionen.

¿Qué elementos podemos utilizar en el depurador?

Existen al menos tres herramientas fundamentales que podemos utilizar en nuestro debugger o depurador.

Son las siguientes:

- ✓ **Breakpoints o puntos de ruptura:** estos puntos pueden ser determinados por el propio programador a lo largo del código fuente de su aplicación. Un breakpoint es un lugar en el programa en el que la ejecución se detiene. Estos puntos se insertan en una determinada línea del código, entonces el programa se pone en funcionamiento y cuando el flujo de ejecución llega hasta él, la ejecución queda congelada y un puntero indica el lugar en el que la ejecución se ha detenido. El depurador muestra los valores de las variables tal y como están en ese momento de la ejecución. Cualquier discrepancia entre el valor actual y el valor que deberían tener supone una importante información para el proceso de depuración.
- ✓ **Ejecución paso a paso:** el depurador también nos permite ejecutar un programa paso a paso, es decir, línea por línea. A través de esta herramienta podremos seguir el progreso de ejecución de nuestra aplicación y supervisar su funcionamiento. Cuando la ejecución no es la esperada quizá estemos cerca de localizar un error o bug. En ocasiones, si utilizamos métodos procedentes de la biblioteca estándar no necesitaremos hacer un recorrido paso a paso por el interior de estos métodos, ya que es seguro que no contendrán errores internos y podremos ahorrar tiempo no entrando en su interior paso a paso. El debugger ofrece la posibilidad de entrar o no en dichos métodos.
- ✓ **Observación de variables y atributos en la ejecución paso a paso**: una de las mayores ventajas que ofrecen la mayoría de los depuradores es la posibilidad de observar (e incluso manipular) el valor de las variables en tiempo real durante la ejecución, así como el resultado de la evaluación de expresiones o subexpresiones que forman parte de una sentencia.

Depurando con Netbeans