Uso de estructuras de control.

Caso práctico

Los miembros de BK Programación están inmersos en el mundo de Java, ya conocen una buena parte del lenguaje y ahora van a empezar a controlar el comportamiento de sus programas.

María pregunta a **Juan**:

-Dime Juan, ¿las estructuras de control en Java son similares a las de cualquier otro lenguaje?

-Efectivamente, **María**, como la gran mayoría de los lenguajes de programación, Java incorpora estructuras que nos permiten tomar decisiones, repetir código, etc. Cada estructura tiene sus ventajas, inconvenientes y situaciones para las que es adecuada. Hay que saber dónde utilizar cada una de ellas -aclara **Juan**.

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

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). Es decir, 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.

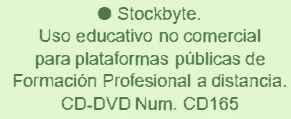
Además de las sentencias típicas de control de flujo, en esta unidad haremos una revisión de las **sentencias de salto incondicional**, que aunque **son altamente desaconsejables en la mayoría de casos, y generalmente resulta innecesario recurrir a ellas**, no está de más conocerlas por si te las encuentras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

Vamos entonces a ponernos el mono de trabajo y a coger nuestra caja de herramientas, ¡a ver si no nos mojamos mucho!

2.- Estructura secuencial: sentencias y bloques.

Caso práctico



Ada valora muy positivamente en un programador el orden y la pulcritud.

Organizar correctamente el código fuente es de vital importancia cuando se trabaja en entornos colaborativos en los que son varios los desarrolladores que forman los equipos de programación. Por ello, incide en la necesidad de recordar a **Juan** y **María** las nociones básicas a la hora de escribir programas.

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?

Bloques de sentencias.

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?**
  Bueno... 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

Accede a los tres archivos que te ofrecemos a continuación y compara su código fuente. Verás que los tres obtienen el mismo resultado, pero la organización de las sentencias que los componen es diferente entre ellos.

Sentencias, bloques y diferentes organizaciones.

Bloque de sentencias para ejecución secuencial.	Bloque de sentencias con declaración de variables.	Bloque de sent código organizad de variables, ent procesamier

int dia=15;
System.out.println("El día es: "+dia);
int mes=9;
System.out.println("El mes es: "+mes);
int anio=2023;
System.out.println("El anio es: "+anio)
	// Zona de declaración de variables
	int dia=10;
	int mes=11;
	int anio=2011;
	
	// Sentencias que usan esas variables
	System.out.println("El día es: "+dia);
	System.out.println("El mes es: "+mes);
	System.out.println("El año es: "+anio);
		// Zona de declaración de
		int dia;
		int mes;
		int anio;
		String fecha;
		
		//Zona de inicialización<b
		dia=10;
		mes=11;
		anio=2011;
		fecha="";
		
		//Zona de procesamiento
		fecha=dia+"/"+mes+"/"+a
		
		//Zona de salida de resulta
		System.out.println ("La fe

En este primer ejemplo, las sentencias están colocadas en orden secuencial. Sentencias en orden secuencial. (0.01 MB)	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. Sentencias y declaraciones de variables. (0.01 MB)	En este tercer archi apreciar que se ha código en las siguie declaración de varia datos de entrada, p dichos datos y obte salida. Este tipo de más estandarizada nuestros programas legibilidad y claridad Sentencias, organización del

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**.

Autoevaluación

Indica qué afirmación es la más correcta:

- ○ Para crear un bloque de sentencias, es necesario delimitar éstas entre llaves. Este bloque funcionará como si hubiéramos colocado una única orden.
- ○ La sentencia nula en Java, se puede representar con un punto y coma sólo en una única línea.
- ○ Para finalizar en Java cualquier sentencia, es necesario hacerlo con un punto y coma.
- ○ Todas las afirmaciones son correctas.

Recomendación

Este es un buen momento para que vuelvas al apartado de "organización" del código que vimos en la unidad anterior y recuperes la "plantilla estándar de programa principal" que se os proporcionaba para realizar vuestros ejercicios y tareas. La vamos a usar como base para una buena parte del curso, especialmente para las primeras unidades.

2.1.- Ámbito o alcance de una variable.

Gerd Altmann (Pixabay License)

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
        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
            System.out.println();
        }
        System.out.println("En bloque principal (0)");
        System.out.println("num1= " + num1);
        //System.out.println("num2= " + num2);// Error
        //System.out.println("num3= " + num3);// Error
        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.

Mostrar retroalimentación

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);
```

Mostrar retroalimentación

¿Y para el caso de la **línea 48**?

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

Mostrar retroalimentación

¿Y respecto a la línea 53?

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

Mostrar retroalimentación

3.- Estructuras de selección o condicionales.

Caso práctico

Juan está desarrollando un método en el que ha de comparar los valores de las entradas de un usuario y una contraseña introducidas desde el teclado, con los valores almacenados en una base de datos. Para poder hacer dicha comparación necesitará utilizar una estructura condicional que le permita llevar a cabo esta operación, incluso necesitará que dicha estructura condicional sea capaz de decidir qué hacer en función de si ambos valores son correctos o no.

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? Para comenzar, 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.

Curiosidad

En el lenguaje de programación C, verdadero o falso se representan mediante un literal entero. 0 representará Falso y 1 o cualquier otro valor, representará Verdadero. Como sabes, en Java las variables de tipo ___booleano sólo podrán tomar los valores true (verdadero) o false (falso).

La evaluación de las condiciones o expresiones que controlan las estructuras de selección, devolverán siempre un valor verdadero o falso.

Las estructuras de selección se dividen en:

Marcin Wichary

- 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**.

A continuación, detallaremos las características y funcionamiento de cada una de ellas. Es importante que a través de los ejemplos que vamos a ver, puedas determinar en qué circunstancias utilizar cada una de estas estructuras. Aunque un mismo problema puede ser resuelto con diferentes estructuras, e incluso, con diferentes combinaciones de éstas.

3.1.- Estructura condicional simple: 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 if en lenguaje natural

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 en Java

Estructura if simple.	
Sintaxis para el caso de ejecución condicional de una sola rama con una sola sentencia	**Sintaxis para el caso de ejecución condicional de una sola rama con un bloque de sentencias**

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

```
int valor = -10;
System.out.println ("El programa está ejecutándose."); // Esta línea se ejecuta en cualquier caso
if (valor
```

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:

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

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:

```
int valor = -10;
System.out.println ("El programa está ejecutándose."); // Esta línea se ejecuta en cualquier caso
if (valor
```

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:

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

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 sea <code>true, el código tendría el siguiente aspecto:

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

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 haya sido <code>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:

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

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

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**?

Mostrar retroalimentación

Reflexiona

Recuerda que la línea de código:

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

aún no somos capaces de interpretarla completamente pero sabemos que es **necesaria para poder leer datos desde el teclado** posteriormente usando esta otra línea:

```
numero = teclado.nextInt();
```

Así lo llevamos haciendo desde la unidad anterior aunque todavía no sepamos qué se está haciendo en realidad. En la siguiente unidad aprenderemos mejor cómo trabajar con la entrada desde teclado. Mientras tanto sabemos que podemos hacerlo así:

1. **declarando e instanciando un objeto de tipo** Scanner (un tipo referenciado que estudiaremos más adelante): la variable que hemos llamado teclado (**Scanner teclado = new Scanner(System.in)**);
2. **usando ese objeto para leer una entrada** desde teclado usando alguna de las sentencias teclado.nextInt(), **teclado.nextDouble()**, teclado.nextShort(), etc. según queramos leer un int, **double**, short, **float**, etc. En este caso se ha efectuado la lectura de un entero y lo leído se ha almacenado en una variable de tipo entero: numero= teclado.nextInt().

En la próxima unidad ya entenderemos qué significa instanciar un objeto o llamar a u método. Por ahora nos conformaremos con utilizar este mecanismo sin preocuparnos especialmente de su funcionamiento interno.

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 if-else en lenguaje natural

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-else en Java

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.

Mostrar retroalimentación

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 aterior 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**.

Autoevaluación

¿Qué mostrará por pantalla el siguiente fragmento de código?

```
int x=100;
if ( x
```

Sugerencia

- Mostrará el texto "¡Enhorabuena!".
- No se mostrará nada por pantalla.
- Mostrará el texto "¡Lo siento!".

¿Qué mostrará por pantalla el siguiente fragmento de código?

```
int x=100;
if ( x &gt;= 100 )
   System.out.println ("¡Enhorabuena!");
else
   System.out.println ("¡Lo siento!");
```

- Mostrará el texto "¡Enhorabuena!".
- No se mostrará nada por pantalla.
- Mostrará el texto "¡Lo siento!".

¿Qué mostrará por pantalla el siguiente fragmento de código?

```
int x=100;
if ( x
```

- Mostrará el texto "¡Enhorabuena!".
- No se mostrará nada por pantalla.
- Mostrará el texto "¡Lo siento!".

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

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**.

Mostrar retroalimentación

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** o bien **!(numero > 0)**.

Mostrar retroalimentación

¿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.

Mostrar retroalimentación

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

Mostrar retroalimentación

Ejercicio Resuelto

Nos han pedido que escribamos un programa en Java que a partir de una calificación (número real) entre 0 y 10 (ambos inclusive) muestre por pantalla el

calificación (número real) entre 0 y 10 (ambos inclusive) muestre por pantalla el mensaje "*APROBADO*" si la nota es igual o superior a 5 o bien "*SUSPENSO*" si es inferior a 5.

Implementa ese programa utilizando una **estructura de tipo condicional**. Intenta que el texto de resultado sea almacenado en una variable de salida (llamada por ejemplo calificacionCualitativa) y procura separar el programa en los tres bloques que recomendábamos en la unidad anterior: **entrada de datos**, **procesamiento** y **salida de resultados**.

Coloca un comentario delante de cada bloque y **separa cada bloque con al menos una línea en blanco**. Coloca también **un primer bloque adicional donde declares todas las variables** que vayas a utilizar incluyendo a su derecha un pequeño comentario sobre cada variable. Observarás que el código queda mucho más claro y legible.

Mostrar retroalimentación

Dado que en nuestra escuela cualquier valor por debajo de 0.0 o por encima de 10.0 no tiene sentido como una calificación numérica, amplía el programa anterior para que si se introduce un valor fuera del rango válido, no se lleve a cabo la comprobación y se muestre un mensaje de salida del tipo "*Calificación introducida no válida*".

Mostrar retroalimentación

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).

Autoevaluación

¿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");
```

Sugerencia

- Nunca.
- Siempre.
- Cuando el resto de la división entre 2 del contenido de la variable numero sea cero.

3.3.- Estructuras condicionales anidadas.

Jae Rue (Pixabay License)

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 &gt; 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 &gt; 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 &gt; 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 &amp;amp;&amp;amp; x &gt; 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  ) {  // 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  )
      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.

Mostrar retroalimentación

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.

Mostrar retroalimentación

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.

Mostrar retroalimentación

Ejercicio Propuesto

Se nos plantea la posibilidad de hacer un programa que calcule la calificación de un examen a partir de las preguntas acertadas, falladas y no contestadas.

Implementa un programa en Java que calcule la nota de un examen de tipo test de **20 preguntas**.

El programa debe solicitar por teclado **dos números enteros**: **número de preguntas acertadas** y **número de preguntas falladas** y a partir de ahí calcular la nota final teniendo en cuenta que **cada fallo restará la mitad de un acierto**, y que obviamente las **preguntas sin contestar ni restarán ni sumarán**. La nota final deberá estar entre 0 y 10. Si la suma del número de preguntas acertadas y el número de preguntas falladas supera el total de preguntas (20), entonces no se llevará a cabo ningún cálculo y se mostrará por pantalla el mensaje "*Datos erróneos*". Si los datos son correctos deberán entonces calcularse la calificación no numérica del examen y mostrarse por pantalla ambas calificaciones: la cuantitativa (numérica) y la cualtiativa (no numérica).

Mostrar retroalimentación

3.4.- Estructura selectiva múltiple: switch (I).

¿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`.

Autoevaluación

Indica para cada afirmación si es verdadera o falsa.

La estructura tipo switch resulta imprescindible en cualquier lenguaje, ya que de lo contrario no se podrían tomar decisiones que implican la posibilidad de seguir por más de dos caminos.

○ Verdadero ○ Falso

La estructura tipo switch es más eficiente para resolver un problema con múltiples posibilidades, que el uso de estructuras if/else anidadas para resolver ese mismo problema.

○ Verdadero ○ Falso

¿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.

Estructura selectiva múltiple en Jav

Sintaxis	Condicio

```
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 la tipo compatible. Sólo desde la versión 7 del lenguaje Java se p 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 pu
- El bloque de sentencias asociado a la cláusula **default** puede diferencia de funcionamiento. Se permite por mantener la estru necesita si el **default** va al final, tal y como se ve en el código,

Funcionamiento:

- Las diferentes alternativas de esta estructura estarán precedidas de la cláusula **case** que el valor obtenido al evaluar la expresión del **switch**.
- Tradicionalmente, en las cláusulas **case** de Java **no podían indicarse expresiones con** otros lenguajes de programación sí lo permiten). Habrá que asociar una cláusula **case** a Esto ha cambiado desde el JDK 12, que sí permite especificar listas de valores a través
- Se pueden especificar varios **case** uno detrás de otro, de manera que para todos esos va
- La cláusula **default** será utilizada para indicar un caso por omisión o por defecto (cualqui realidad). Las sentencias asociadas a la cláusula **default** se ejecutarán si ninguno de los 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 finalizar
- Cada cláusula **case** puede llevar asociadas una o varias sentencias, sin necesidad de de
- En el momento en el que el resultado de la evaluación de la expresión coincide con algu ejecutarán todas las instrucciones asociadas hasta la aparición de una sentencia **break** posteriores). Esto hace que si en una cláusula **case** no se incluye un **break** para finalizar **cascada"**, de forma que seguirá ejecutando las sentencias de la siguiente cláusula **case** en alguna encuentre una sentencia **break**, o hasta que alcance el final de la sentencia. E sentencia **switch** y hacer algo parecido a los rangos que sí permiten otros lenguajes, au

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 de uso de la estructura switch en Java

Ejemplo resuelto con switch

```
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");
}
```

Ejemplo resuelto con if-else anidados

```
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;

Mostrar retroalimentación

3.4.1.- Estructura selectiva múltiple: switch (II).

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) {<br>
  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:

```
switch (expresion) {

  case valor1: case valor2:
    sentencial_1;
    sentencial_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.

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.).

Recomendación

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.

Autoevaluación

Para la estructura tipo switch, el último caso que debe incluirse es siempre la etiqueta default, ya que debe indicarse expresamente al compilador qué debe hacer ante cualquier valor resultante de evaluar la expresión distinto de los contemplados en las etiquetas anteriores. De lo contrario, se produciría una indefinición, que provocaría que el programa abortara si se presentara cualquier otro caso no previsto expresamente con alguna etiqueta asociada a las instrucciones a ejecutar.

○ Verdadero ○ Falso

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.

Mostrar retroalimentación

Para saber más

Una explicación bastante buena de la sentencia switch en Java puedes encontrarla en el siguiente enlace:

Sentencia switch en Java

Una web con algunos ejemplos típicos de uso de la sentencia switch en Java:

Ejemplos de ejercicios resueltos en Java usando if y switch

4.- Estructuras repetitivas, iterativas o cíclicas.

Caso práctico

Juan ya tiene claro cómo realizar la comprobación de los valores de usuario y contraseña introducidos por teclado, pero le surge una duda:

¿Cómo podría controlar el número de veces que el usuario ha introducido mal la contraseña?

Ada le indica que podría utilizar una estructura de repetición que solicitase al usuario la introducción de la contraseña hasta un máximo de tres veces. Aunque comenta que puede haber múltiples soluciones y todas válidas, lo importante es conocer las herramientas que podemos emplear y saber cuándo aplicarlas.

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 algunos de estos bucles son equivalentes entre sí 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 (I).

En la siguiente imagen puedes ver un diagrama de flujo que representa el funcionamiento de este tipo de estructura repetitiva.

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 podría 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.

Autoevaluación

De las afirmaciones siguientes, marca las que te parezcan correctas.

- [ ] Cualquier problema que se pueda resolver con estructuras tipo for también es posible resolverlo con estructuras tipo while.
- [ ] Los ciclos while son adecuados para aquellos casos en los que sabemos que las instrucciones del cuerpo del bucle se deben ejecutar al menos una vez.
- [ ] Para casos en los que sabemos exactamente cuántas veces debe ejecutarse el bloque de sentencias contenido del ciclo, while es mucho más adecuado que for, porque aunque hace lo mismo, permite presentar el código de una manera más simple y clara.
- [ ] Los ciclos while requieren que dentro de las sentencias del bucle haya alguna que modifique el valor de la condición que controla el bucle.

Mostrar retroalimentación

¿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.

Estructura repetitiva while.

Estructura repetitiva while.

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
```

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
```

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

```
int numero=1;
while ( 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
```

Mostrar retroalimentación

4.1.1.- Estructura repetitiva while (II).

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 &gt; 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 &gt; 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.

Ejercicio Resuelto

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

Mostrar retroalimentación

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.

Recomendación

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

Autoevaluación

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

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

¡IMPORTANTE! Lee con atención la retroalimentación, porque en ella se explican algunos aspectos muy a tener en cuenta sobre la construcción de bucles con este tipo de condiciones.

- ○ Verdadero
- ○ Falso

4.2.- Estructura repetitiva do-while (I).

En la siguiente imagen puedes ver un diagrama de flujo que representa el funcionamiento de este tipo de estructura repetitiva.

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 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.

Autoevaluación

Indica cuál es la estructura de control de flujo repetitiva o cíclica que garantiza que las sentencias del cuerpo del bucle se ejecutarán al menos una primera vez, con independencia del valor de la condición en el momento de comenzar la ejecución del ciclo.

- ○ do-while.
- ○ for.
- ○ while.
- ○ for/in.
- ○ switch.

¿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.

Estructura repetitiva do-while.

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 siguiente se ejecuta al menos la primera vez
```

```
do { // El cuerpo del bucle siempre se ejecuta al menos la primera vez
   System.out.println (numero);
   numero++;
} while ( numero
```

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

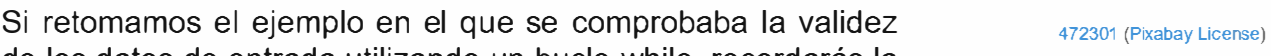
```
numero=1;
do
   System.out.println (numero++);
while ( numero
```

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 un bucle while. En este caso debes utilizar un bucle do-while.

Mostrar retroalimentación

4.2.1.- Estructura repetitiva do-while (II)



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 bucl
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.

Mostrar retroalimentación

Autoevaluación

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

- La necesidad de duplicar el código de las sentencias de control del bucle justo antes de entrar al mismo, de forma que se garantice así que al menos se van a ejecutar una vez.
- 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.
- La posibilidad de que el bucle no se ejecute nunca, por lo que la sentencia que modifique la variable de control del bucle debe ser previa al do-while, o de lo contrario estaríamos dejando la puerta abierta a que nunca se modificara esa sentencia, y por tanto, nunca se alcanzara la condición de salida, produciendo un bucle infinito, que es algo que siempre hay que evitar.
- Las respuestas anteriores son todas incorrectas.

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.

Mostrar retroalimentación

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

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.

○ Verdadero ○ Falso

4.3.- Concepto de contador.

Clker-Free-Vector-Images (Pixabay License)

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 &amp;amp;&amp;amp; numIntentos
```

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, que mientras sea <code>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:

```
// 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 &amp;amp;&amp;amp; numIntentos
```

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.
```

Mostrar retroalimentación

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.

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
```

Mostrar retroalimentación

4.4.- Concepto de acumulador.

PublicDomainArchive (Pixabay License)

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
```

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
```

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.
```

Mostrar retroalimentación

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.).

Por otro lado, a veces en lugar de hacer más grande el valor del acumulador, se hace más pequeño (sería un "desacumulador") pues en lugar de restar se suma o en lugar de multiplicar se divide.

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.
```

Mostrar retroalimentación

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.
```

Mostrar retroalimentación

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.
```

Mostrar retroalimentación

4.5.- Estructura repetitiva for (I).

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.

Autoevaluación

Rellena los huecos con la palabra adecuada.

En la estructura repetitiva tipo for lo primero que debe hacerse es una correcta [ ] de la variable contador que controla la ejecución del bucle, luego se [ ] el valor de la variable para comprobar si se ha alcanzado el número de [ ] que deben realizarse, o si por el contrario debe realizar una nueva [ ] . Finalmente, como último paso de cada iteración, se [ ] el valor de la variable [ ] mediante sentencias de [ ] o [ ] .Es importante que el contador se actualice modificando su valor para que en algún momento se alcance la [ ] de [ ] del bucle, ya que de lo contrario podríamos caer en un [ ] [ ] ,quedebe evitarse a toda costa.

Enviar

En la siguiente tabla, podemos ver la especificación para el lenguaje Java de la estructura for:

Estructura repetitiva for.

Sintaxis	Funcionamiento

Estructura for con una única sentencia (no necesita las llaves):
 	
`for (inicialización ; condición ; incremento)`
`  sentencia;`
	✓ 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

Estructura for con un bloque de sentencias:
 	
`for (inicialización ; condición ; incremento) {`
`  sentencia_1;`
`  sentencia_2;`
`  ...`
`  sentencia_N;`
`}`	

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) y el incremento (<code>numero++) los tenemos en la propia cabecera de la estructura `for`:

```
for ( numero=1 ; 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
```

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
```

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
```

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
```

Mostrar retroalimentación

4.5.1.- Estructura repetitiva for (II).

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
```

2. Condiciones complejas donde se usan operadores lógicos:

```
for (contador1=0 ; contador1
```

3. Incremento o decremento de más de un contador:

```
for (contador1=0, contador2=0; contador1
```

4. Ausencia de inicialización:

```
for ( ; contador1
```

5. Ausencia de inicializaciones y de incrementos o decrementos:

```
for ( ; contador1
```

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.

Mostrar retroalimentación

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.

Autoevaluación

Cuando construimos la cabecera de un bucle for, podemos prescindir de alguno de los tres elementos que la forman e incluso, podemos utilizar más de una variable contadora separando éstas por comas. Pero, ¿qué conseguiremos si construimos un bucle de la siguiente forma?

```
for ( ; ; ){ //instrucciones }
```

- ○ Un bucle infinito.
- ○ Nada, dará un error.
- ○ Un bucle que se ejecutaría una única vez.

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 en lugar de <code>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
- 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
```

Mostrar retroalimentación

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.
```

Mostrar retroalimentación

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 proceso de cálculo que consiste en ir diviendo entre 10 hasta que se obtenga cero como cociente. 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.
```

Mostrar retroalimentación

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.

Caso práctico

Juan recuerda que algunos lenguajes de programación permitían realizar saltos a lo largo de la ejecución de los programas, incluso a zonas remotas del código, saltándose los ámbitos de las estructuras de control de flujo incluso, y conoce algunas sentencias que aún se siguen utilizando para ello. Le pregunta a **Ada**:

-¿Es posible prescindir por completo de las estructuras de salto incondicional? ¿Es posible programar cualquier salto incondicional de forma que se prescinda de él, y sólo se hagan los saltos ligados a las estructuras de control condicionales o cíclicas, tras comprobar sus condiciones?

Ada, mientras toma un libro sobre Java de la estantería del despacho, le aconseja:

Las instrucciones de salto incondicional a veces han sido mal valoradas por la comunidad de programadores, pero en Java algunas de ellas son totalmente necesarias. También es verdad que las situaciones en las que resultan o bien útiles o bien necesarias están muy delimitadas, y que usarlas fuera de esos casos concretos es en general evitable, y además resulta muy desaconsejable, ya que produce código menos claro, difícil de entender y de mantener, y por tanto más costoso, así que yo te voy a pedir que tengas muy claro cuándo debes usarlas y cuándo debes evitarlas, porque en esta empresa queremos hacer software de calidad a buen precio. Mira, en este libro se habla del uso de las sentencias break, `continue` y return en Java.

-Vale, voy a empaparme del asunto. Gracias por el libro.

¿Saltar o no saltar? He ahí la cuestión.

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
```

¡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.

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:

```
/**
 * 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
```

¡¡Recuerda!!

Con la sentencia continue, también se desaconseja el uso. Igual que pasaba con el **una mala práctica** de programación, que esconde un mal diseño de la lógica asociada al ciclo 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.

Para clarificar algo más el funcionamiento de ambas sentencias de salto, vuelve a mirar detenidamente el diagrama representativo del comienzo de este epígrafe.

Reflexiona

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

Mostrar retroalimentación

Autoevaluación

La instrucción break puede utilizarse en las estructuras de control switch, while, for y do-while, pudiendo omitirse en la cláusula default de la estructura switch.

- ○ Verdadero
- ○ Falso

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.

Mostrar retroalimentación

5.2.- Etiquetas.

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

Abrev

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

No obstante, 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 indeseable y evitable **debería ser necesario recurrir a etiquetas.**

¿Y cómo se crea un salto a una etiqueta?

En primer lugar, 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></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
```

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.

Mostrar retroalimentación

6.- Errores, pruebas y depuración de programas.

Caso práctico

Ada y **Juan** ya conocen las capacidades del depurador que incorpora el entorno NetBeans y van a enseñar a **María** las ventajas de utilizarlo.

-Puedes depurar tus programas haciendo dos cosas: creando **breakpoints** o haciendo ejecuciones paso a paso-, le comenta **Juan**.

María, que estaba codificando un nuevo método, se detiene un momento y pregunta:

-Entonces, ¿cuando el programa llega al breakpoint podré saber qué valor tiene una variable determinada?

-Efectivamente, **María**, y podrás saber el valor de aquellas que tú decidas. De este modo a través de los puntos de ruptura y de las ejecuciones paso a paso podrás descubrir dónde puede haber errores en tus programas. Conocer bien las herramientas que el depurador nos ofrece es algo que puede ahorrarnos mucho trabajo-, aporta **Ada.**

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 utilidades (clases, métodos, funciones, etc.) creadas específicamente para ese programa junto con otras que forman parte de bibliotecas o librerías que ya existían previamente. En el caso de Java, estas utilidades son enlazadas (*linked*) solo cuando son llamadas, durante el proceso de ejecución. Pero cuando el programa es compilado, se realizan comprobaciones para saber si esas utilidades 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 instrucción (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 indicando su nombre erróneamente (por ejemplo si escribes System.out.prtln(), que no existe, en lugar de System.out.println(), que sí existe).
- **Sintáctico**: estructuración errónea del código sin seguir las reglas del lenguaje. Algunos ejemplos podrían ser:
  - no cerrar apropiadamente todos los paréntesis que se han abierto en una expresión aritmética;
  - no cerrar apropiadamente 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 no permite la compilación y 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 y aunque el programa compile no puede llegar a generarse un archivo binario ejecutable final.

Durante la fase de ejecución, una vez que el programa ha podido ser compilado, enlazado y ejcutándose. Pueden producirse esencialmente dos tipos de errores durante su funcionamiento:

- **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 "virtual" 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 aunque nos hemos confundido en la redacción de nuestro código, 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. Ejemplos típicos de este tipo de fallos pueden ser:
  - uso de un operador erróneo al realizar algún cálculo o llevar a cabo una comprobación. Por ejemplo, haber escrito el símbolo de suma (+) en lugar de el de multiplicación (*) a la hora de calcular algún resultado, o hacer una comparación usando el operador cuando en realidad lo que necesitábamos usar era el <code>;
  - uso de una variable diferente a la que realmente hace falta al realizar cálculos o llevar a cabo comprobaciones. Por ejemplo, no usar la variable apropiada para comprobar si se debe continuar con un bucle o no;
  - en casos en los que un bloque de código ser ejecutado repetidamente (bucle), colocar alguna de sus sentencias fuera del bucle, dando lugar a una ejecución correcta pero con resultados totalmente diferentes a los esperados.

Es para este último tipo de errores (errores "lógicos" o de la "lógica del programa") 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 las herramientas nos indican 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. En muchos casos incluso el error que indique la máquina virtual de Java (o el sistema operativo si se trata de un ejecutable binario "puro") nos dará la pista del fallo.

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 se trata de 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 producirse y detectarse esos fallos, en algunos casos con la aplicación funcionando ya en producción.

Para descubrir la existencia de este tipo de errores habrá que llevar a cabo una serie de **pruebas sistemáticas** sobre el programa. Estas pruebas se encargarán de comprobar si para cada una de las entradas que se introducen se producen las salidas esperadas. Habría por tanto que preparar una serie de casos de prueba, ejecutarlos y observar los resultados obtenidos. Si algunos de esos resultados no coinciden con lo obtenido, habremos descubierto un fallo en el programa. A partir de ese momento habrá que comenzar a buscarlo.

La **depuración de los programas** nos permitirá 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 habitual que suceda así. A nadie le funciona un programa "a la primera" .

Reflexiona: ¿cómo se prueba un programa?

Para realizar una adecuada **prueba de un programa** se requiere la **confección de casos de prueba** que permitan llevar a cabo una **comprobación sistemática de todas sus opciones y posibilidades**. Es necesario asegurarse de que todas las acciones del programa se llevan a cabo tal y como eran de esperar, es decir, tal y como están descritas en los documentos de análisis y especificación de la aplicación. Si el comportamiento del programa no es así, es que hay algo que no ha sido implementado correctamente por el equipo de desarrollo (los programadores).

Este tipo de trabajo es hoy día una disciplina en sí misma y existen personas especializadas en realizar esta labor. No la suelen realizar los propios programadores, salvo que se trate de proyectos pequeños. Existen para ello los llamados "equipos de prueba" y forman parte del equipo de desarrollo, aunque no se dedican a programar. El trabajo de estas personas consiste en diseñar pruebas sistemáticas de manera independiente y paralela al trabajo de los programadores. Para ello se basarán en los documentos en los que se especifica cuál debe ser el comportamiento del programa. Es decir, tanto programadores como probadores parten del mismo material común: las especificaciones de la aplicación. Sin embargo unos se dedicarán a implementar código y otros a diseñar cómo probar posteriormente ese código.

Una vez codificados los programas, el equipo de pruebas realiza las comprobaciones sistemáticas indicadas en los documentos (u otras herramientas de trabajo) que se hayan confeccionado y se descubran los posibles errores que se hayan podido cometer durante la fase de implementación. Esa información obtenida tras las pruebas es proporcionada al equipo de desarrollo para que localicen en qué parte de su código se han podido producir esos errores y, una vez encontrados, puedan ser corregidos. Es aquí donde entrará el juego el depurador para ayudar a los programadores a localizar los posibles fallos en la implementación.

En este módulo (*Programación*) no se estudia cómo ha de llevarse a cabo este proceso de diseño (y posterior ejecución) de esas pruebas sistemáticas. Se ve algo sobre este tema en el módulo *Entornos de Desarollo*. En nuestro caso, nos conformaremos con "preparar" algunas pruebas genéricas para asegurarnos nosotros mismos que nuestros pequeños programas funcionan correctamente. En el caso de las tareas y otras actividades, normalmente se os proporcionarán algunos "ejemplos de ejecución" o "casos de prueba" suficientes para que podáis probar vuestros programas y aseguraros de que están funcionando tal y como deberían hacerlo. **Recuerda siempre que el hecho de que un programa se ejecute no significa necesariamente que esté haciendo lo que nosotros queríamos que hiciera**.

6.1.- Depuración de código.

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 que estemos utilizando. Este depurador será una herramienta que nos ayudará a eliminar posibles errores de nuestro programa. En nuestro caso concreto se tratará de un entorno de desarrollo para el lenguaje Java. En tal caso 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). Los depuradores suelen mostrar los siguientes elementos simultáneamente en pantalla:

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

A partir de ahí, podremos realizar un análisis y diagnóstico del funcionamiento de nuestro programa mediante el uso de las herramientas que nos proporcionan los depuradores.

¿Cuáles son estas herramientas? 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 (normalmente una barra en un color diferente, por ejemplo rojo) indica el lugar en el que la ejecución se ha detenido. En ese momento es como si "congeláramos el tiempo" y podríamos observar los valores de las variables tal y como están en ese instante. 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 dicho métodos.
- **Observación y manipulación 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. Para ello dispondrás de diversos mecanismos para observar esos valores.

6.2.- Depurando código Java con Netbeans.

A continuación tienes un breve resumen descriptivo de las opciones más habituales que se pueden llevar a cabo durante un proceso de depuración básico con Netbeans.

Para depurar un archivo o un proyecto, puedes usar el menú de depuración o bien el menú contextual de cada pestaña de código Java. Imagina que tienes abierto el archivo "Ejercicio1.java", el menú general de depuración podría aparecer así (puede haber variaciones dependiendo de la versión de Netbeans que se utilice):

Menú del depurador

Para comenzar a depurar podemos elegir una de estas dos opciones:

- ***Debug Project*** o ***Depurar Proyecto*** iniciará la depuración del proyecto y se detendrá cuando encuentre un punto de ruptura. En todo proyecto hay un archivo .java principal, que será el que se depurará al seleccionar esta opción.
- ***Debug File*** iniciará la depuración de un archivo .java concreto, concretamente el que estemos editando en ese momento, deteniéndose en cuanto encuentre un punto de ruptura. Esta será la opción que usaremos en nuestro caso.

Para establecer un nuevo **punto de ruptura** (***breakpoint***) lo más sencillo es hacer clic en sobre el número de línea, aunque también puedes usar la opción de menú ***Toggle Line Breakpoint*** o su atajo de teclado equivalente:

Punto de ruptura.

Una vez que se inicia la depuración y/o se alcanza un punto de ruptura, la opción de **"Step Over (F8)"** o **"Continuar Ejecución (F8)"** está disponible. Esta opción permitirá ir ejecutando el código sentencia a sentencia, sin entrar en ningún método que se pudiera invocar. La opción **"Step Into (F7)"** o **"Paso a paso (F7)"** sí que permitiría ejecutar paso a paso cualquier método que se invoque, entrando dentro del código del método en cuestión (eso lo veremos más adelante, cuando aprendamos a implementar clases y métodos). Pero para el propósito de la tarea de esta unidad, de momento, es suficiente con **"Step Over (F8)"**:

Menú del depurador cuando estamos detenidos en un punto de ruptura.

Por último, para observar el valor de las variables en un momento concreto de la ejecución tenemos la ventana **"*Variables*"** de NetBeans. Si esta ventana no te aparece puedes mostrarla accediendo al menú "***Window > Debugging > Variables***":

Mostrar la ventana de "Variables".

Debes conocer

Para completar tus conocimientos sobre la depuración de programas, te proponemos los siguientes enlaces en los que podrás encontrar cómo se llevan a cabo las tareas básicas de depuración a través del IDE NetBeans.

Depuración básica en NetBeans.

Uso básico del depurador en NetBeans.

A continuación te mostramos un vídeo introductorio a la depuración de programas.

Depurando con NetBeans.

Resumen textual alternativo para Debugging avanzado en NetBeans.

Para saber más

Si deseas conocer algo más sobre depuración de programas, pero a un nivel algo más avanzado, puedes ver el siguiente vídeo.

Debugging avanzado en NetBeans.

Resumen textual alternativo para Debugging avanzado en NetBeans.

Anexo I.- Ejercicios resueltos

Caso práctico

StartupStockPhotos (Licencia Pixabay)

María y Juan han estado trabajando con las diferentes **estructuras de control disponibles en Java**: if-else, **switch**, while, **do-while**, for, etc. así como con las **variables**, **tipos de datos** y **operadores** que vimos en la unidad anterior. Además también ha seguido **leyendo datos desde el teclado** y **mostrando información por pantalla**. Combinando todas esas herramientas ya pueden empezar a construir pequeños programas en Java que reciban cierta información de entrada, la procesen y tomen decisiones en función de esas entradas dando lugar a la ejecución de unas u otras partes del código.

Aún siguen siendo pocas herramientas para trabajar, pero ya son más que suficientes como para plantear soluciones a pequeños problemas en los que haya que automatizar tareas repetitivas o donde sea necesario realizar unos cálculos u otros en función de las entradas recibidas. Para ello van a seguir utilizando la **plantilla genérica de programa en Java** que se os proporcionó en la unidad anterior. Te sugerimos que tú también lo sigas haciendo, tus programas estarán más claros y organizados.

Intercambio de variables

Escribe un programa en Java que lea dos valores reales a y b desde teclado e intercambie esos valores de a y b si el contenido de a fuera mayor que el de b. En caso contrario no se llevará a cabo el intercambio.

StockSnap (Pixabay License)

Aquí tienes un ejemplo de una posible ejecución del programa:

```
INTERCAMBIO DE VALORES
----------
Introduce un valor real para la variable a: 12
Introduce un valor real para la variable b:6
Inicialmente, los valores de las variables son:
a = 12.0, b = 6.0.
Tras el intercambio, los valores de las variables son:
a = 6.0, b = 12.0.
```

Mostrar retroalimentación

Índice de masa corporal

El índice de masa corporal (IMC) se define como el peso en kilogramos dividido por el cuadrado de la altura en metros.

Diagnóstico según IMC

Valor de IMC	Diagnóstico
	Criterio de ingreso en hospital.
de 16 a 17	Infrapeso.
de 17 a 18	Bajo peso.
de 18 a 25	Peso normal (saludable).
de 25 a 30	Sobrepeso (obesidad de grado I).
de 30 a 35	Sobrepeso crónico (obesidad de grado II).
de 35 a 40	Obesidad premórbida (obesidad de grado III).
>40	Obesidad mórbida (obesidad de grado IV).

Escribe un programa en Java que calcule el índice de masa corporal (IMC) de una persona e indique el estado en el que se encuentra esa persona en función del valor de IMC.

Aquí tienes un ejemplo de una posible ejecución del programa:

```
CÁLCULO DEL ÍNDICE DE MASA CORPORAL (IMC)
-----------------------------------------
Introduce el peso (en kg): 97
Introduce la altura (en cm): 182
Para un peso de 97.0 kilogramos y una altura de 1.82 metros:
El índice de masa corporal es de: 29.283902910276534
Tiene sobrepeso de grado I.
```

Mostrar retroalimentación

Menú de opciones

Escribe un programa en Java que permita elegir entre dibujar una recta, un punto o un rectángulo, o bien terminar sin hacer nada.

Para ello, debe mostrarnos un menú en el que nos proporcione las opciones disponibles, y cuya salida sea simplemente escribir por pantalla cuál ha sido la opción elegida.

Aquí tienes un ejemplo de una posible ejecución del programa:

```
DIBUJO DE FIGURAS
-----------------
  1. Dibujar Punto.
  2. Dibujar Recta.
  3. Dibujar Rectángulo
  0. Terminar y salir
Introduce una opción de dibujo: 0
Se ha elegido Terminar y Salir.
```

Mostrar retroalimentación

Suma de números

Escribe un programa en Java que calcule la **suma de varios números reales no negativos introducidos por teclado**.

El proceso debe terminar cuando se introduzca un número negativo por teclado, mostrando en ese momento la suma de los números previos e indicando cuántos números se han sumado. En caso de que el primer número introducido por teclado sea un número negativo, la suma será 0.

Este problema debe ser resuelto usando un bucle tipo while.

Aquí tienes un ejemplo de una posible ejecución del programa:

```
SUMA DE NÚMEROS INTRODUCIDOS POR TECLADO
----------------------------------------
Introduce un número real no negativo: 0,5
Introduce un número real no negativo: 4
Introduce un número real no negativo: 0
Introduce un número real no negativo: 1,2
Introduce un número real no negativo: -7
Se han leído 4 enteros no negativos
Su suma es: 5.7
```

Mostrar retroalimentación

Resolver el mismo ejercicio mediante una estructura de tipo do-while.

Mostrar retroalimentación

Producto de números

Escribe un programa en Java que calcule el **producto de varios números enteros positivos introducidos por teclado**.

El proceso debe terminar cuando se introduzca un número negativo o cero por teclado, mostrando en ese momento el producto de los números previos e indicando cuántos números se han multiplicado. En caso de que el primer número introducido por teclado no sea positivo, el producto será 1.

Este problema debe ser resuelto usando un bucle tipo while.

Aquí tienes un ejemplo de una posible ejecución del programa:

```
PRODUCTO DE NÚMEROS INTRODUCIDOS POR TECLADO
--------------------------------------------
Introduce un número entero positivo: 3
Introduce un número entero positivo: 11
Introduce un número entero positivo: 1
Introduce un número entero positivo: 2
Introduce un número entero positivo: 0
Se han leído 4 enteros positivos
Su producto es: 66
```

Mostrar retroalimentación

Resolver el mismo ejercicio mediante una estructura de tipo do-while.

Mostrar retroalimentación

Sumas y medias

Escribe un programa en Java que calcule la **suma** y la **media aritmética** para una serie de **números enteros no negativos** leídos desde teclado. La cantidad de números a leer será lo primero que deberemos solicitar al usuario.

- Si el valor fuera negativo, terminará la ejecución con un mensaje indicando el error: no es posible leer una cantidad negativa de números, y terminará.
- Si el valor fuera 0, nos mandará un mensaje indicando que no se ha leído ningún número, y terminará.
- Si el valor fuera positivo, nos leerá la cantidad de números indicada, y calculará su suma y su media aritmética.
  - Una vez conocida dicha cantidad, se repetirá un ciclo tipo while que se encargará de leer los números necesarios, de manera que si un número es negativo, se descartará y se volverá a leer otro número en su lugar. Así, el programa terminará su ejecución cuando se hayan leído todos los números requeridos. Como salida nos mostrará:
    - El total de intentos de lectura que se han hecho.
    - El total de números correctos (no negativos) que se han leído.
    - La suma de los números no negativos leídos y su media aritmética.

Aquí tienes un ejemplo de una posible ejecución del programa:

```
SUMA Y MEDIA DE NÚMEROS INTRODUCIDOS POR TECLADO
-------------------------------------------------
Leeremos varios números positivos introducidos por teclado
para calcular su suma y su media aritmética
¿Cuántos números no negativos quieres leer? 4
Introduzca número 1º: -1
El número -1 es negativo, y se descarta.
Introduzca número 1º: 2
Introduzca número 2º: 0
Introduzca número 3º: -4
El número -4 es negativo, y se descarta.
Introduzca número 3º: -9
El número -9 es negativo, y se descarta.
Introduzca número 3º: 4
Introduzca número 4º: 6
Se han realizado un total de 7 intentos de lectura de números.
De ellos 4 han sido correctos.
Suma total: 12. Media: 3.0.
FIN DEL PROGRAMA.
```

Mostrar retroalimentación

Resolver el mismo ejercicio mediante una estructura de tipo do-while.

Mostrar retroalimentación

Repetir el ejercicio con la siguiente variación: en este caso **lo que se leerá al comienzo no será la cantidad de números enteros no negativos que deben leerse, sino el total de intentos que deben hacerse**. Al finalizar, se indicará igual que antes los intentos realizados, el total de números correctos (no negativos) que se han leído, la suma de los mismos y su media aritmética. En este caso, al ser un número fijo de intentos a realizar, lo más adecuado es resolverlo usando un bucle for.

Aquí tienes un ejemplo de una posible ejecución del programa:

```
SUMA Y MEDIA DE NÚMEROS INTRODUCIDOS POR TECLADO
--------------------------------------------------
Leeremos varios números positivos introducidos por teclado
para calcular su suma y su media aritmética
¿Cuántos números quieres leer? 5
Introduzca número 1º: -7
El número -7 es negativo, y se descarta.
Introduzca número 2º: 1
Introduzca número 3º: 0
Introduzca número 4º: -2
El número -2 es negativo, y se descarta.
Introduzca número 5º: 5
Se han realizado un total de 5 lecturas de números.
De ellos 3 han sido correctos.
Suma total: 6. Media: 2.0.
FIN DEL PROGRAMA.
```

Mostrar retroalimentación

Clasificación de números

Escribir un programa en Java que lea una serie de 10 números enteros desde teclado. Una vez leídos todos los números se indicará por pantalla cuántos de ellos son positivos, cuántos negativos, y cuántos cero. También se calculará y mostrará la suma de los negativos y su media, la suma de los positivos y su media, y la suma de los 10 y su media.

Aquí tienes un ejemplo de una posible ejecución del programa:

```
CLASIFICACIÓN, SUMA Y MEDIA DE NÚMEROS INTRODUCIDOS POR TECLADO
---------------------------------------------------------------
Leeremos 10 números introducidos por teclado
para clasificarlos, calcular su suma y su media aritmética
Número 1º: 1
Número 2º: 2
Número 3º: 3
Número 4º: 0
Número 5º: -1
Número 6º: 0
Número 7º: -2
Número 8º: 0
Número 9º: 5
Número 10º: -3
En total se han leído:
4 números positivos, 3 números negativos y 3 ceros.
Los positivos suman 11 y su media es 2.75.
Los negativos suman -6 y su media es -2.0
La suma de los 10 números leídos es 5 y su media es 0.5
```

Mostrar retroalimentación

Entrenamiento de natación

David Mark (Licencia Pixabay)

Una deportista entrena en la piscina haciendo un largo de ida a estilo crol, un largo de vuelta a estilo espalda, un largo de ida a estilo braza y de nuevo vuelta a estilo espalda, y así sucesivamente.

Implementar un programa en Java que solicite al usuario la

Implementar un programa en Java que solicite al usuario la **cantidad de largos** que ha hecho la nadadora. Debe estar en el **rango 0-50**, ambos valores incluidos. Si el valor no está en el rango permitido deberá volverse a solicitar para que se introduzca una cantidad válida. Se permitirá introducir hasta tres valores (tres intentos). **Un tercer intento no válido hará que el programa finalice sin calcular nada** indicando que se ha superado el máximo de intentos erróneos. Los **valores límite** del número de largos que se usarán para las comprobaciones (mínimo 0, máximo 50) deben estar almacenados como **variables de tipo constante** en lugar de usar literales. También debe usarse una variable de tipo constante para almacenar el **número máximo de intentos** (tres).

En el caso de que se haya introducido finalmente una cantidad válida se procederá a componer, **utilizando obligatoriamente un bucle**, una **cadena de caracteres** donde se representará el **desarrollo de cada uno de los largos** que ha realizado la nadadora durante su entrenamiento. La cadena, que se mostrará finalmente como resultado de la ejecución del programa, tendrá la siguiente estructura:

1. comenzará con una **apertura de llave** (carácter '{') y un **espacio en blanco**;
2. se irá indicando cómo se ha hecho cada largo alternando las palabras "*Crol*", "*Espalda*", "*Braza*", "*Espalda*", "*Crol*", "*Espalda*", etc. Cada palabra debe ir separada de la anterior y de la siguiente por un espacio;
3. se terminará con un espacio en blanco y el cierre de llave (carácter '}').

Si se introducen cero largos como entrada, simplemente se mostrará una lista vacía de largos entre llaves.

Aquí tienes un ejemplo de ejecución del programa:

Aquí tienes un ejemplo en el que se introducen tres valores inválidos y no llega a calcularse el resultado:

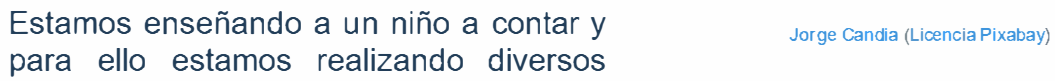
Y aquí algunos casos de prueba adicionales:

Ejemplos de casos de prueba

Largos	Desarrollo del entrenamiento
0	{ }
1	{ Crol }
2	{ Crol Espalda }
3	{ Crol Espalda Braza }
4	{ Crol Espalda Braza Espalda }
5	{ Crol Espalda Braza Espalda Crol }
11	{ Crol Espalda Braza Espalda Crol Espalda Braza Espalda Crol Espalda Braza }

Mostrar retroalimentación

Escalera incremental



Jorge Candia (Licencia Pixabay)

Estamos enseñando a un niño a contar y para ello estamos realizando diversos puzzles y juegos para estimular su curiosidad y creatividad.

Escribir un programa en Java que solicite por teclado un número de filas (entre 1

Escribir un programa en Java que solicite por teclado un número de filas (entre 1 y 10) y que genere una **cadena de caracteres** con una escalera del siguiente tipo: en cada fila habrá una cantidad de números igual al número de filas en las que se esté. Se empezará contando desde el 1 en la primera fila y ese contador se irá "arrastrando" en cada fila sucesiva. Cada número debe ir separado del siguiente por un espacio en blanco. Es decir en la primera fila habrá un único número, el 1, en la segunda dos números (2 y 3), en la tercera tres números (4, 5 y 6) y así sucesivamente 1 tal y como se puede observar en el ejemplo siguiente:

Observa que **al principio de cada fila se debe escribir el número de fila, dos puntos y la secuencia del contador incrementándose**.

Si se introduce una cantidad de filas que no se encuentra en el rango permitido (1-10, ambos inclusive), el programa volverá a solicitar la cantidad de filas hasta que esta sea correcta **el número de veces que sea necesario**. Para llevar a cabo este control te recomendamos utilizar un bucle do-while.

Una vez que se haya construido la cadena con la escalera debes mostrarla por pantalla como resultado final del programa. **Para construir la escalera debes utilizar bucles** for.

Aquí tienes algunos ejemplos de ejecución:

Aquí tienes algunos ejemplos de ejecución:

Recuerda que para que un String incluya un "salto de línea" basta con concatenar (operación +) a esa cadena un carácter de salto de línea ("\n"): cadena += "\n";

Mostrar retroalimentación

Condiciones y términos de uso de los materiales

Historial de actualizaciones

Versión: 02.01.00	Fecha de actualización: 04/11/21	Autoría: Diosdado Sánchez Hernández

Ubicación: Apartado 6.2
Mejora (tipo 1): Respecto a la segunda sugerencia: Apartado 7. Reforzar el uso del depurador, detallando el proceso para depurar código con un vídeo o contenido interactivo. El depurador ya no está en el apartado 7 sino en 6.2 y se incluyen dos vídeos.
Ubicación: 6.- Depuración de programas
Mejora (tipo 2): - Cambio de nombre a 6. Errores, pruebas y depuración de programas.
- Faltaría comentar algunos posibles ejemplos de errores lógicos en el código (confusión de operadores, de variables, etc.) que permiten que el programa se ejecute, pero que en determinadas ocasiones no funcione como debería funcionar. Esto daría paso a poder mencionar la necesidad de pruebas sistemáticas en el código, cosa que se ve en el módulo Entornos de Desarrollo.
- Incluir un iDevice Reflexiona titulado ¿Cómo se prueba un programa?, donde se realice una breve introducción al proceso de pruebas del código y cómo lo enfocaremos a lo largo del módulo (mediante casos de prueba o ejemplos de ejecución en ejercicios y tareas).

6.1.- Depuración de código en Java
- Cambio de nombre a 6.1.- Depuración de código, para simplemente hablar de la utilidad y las funciones principales de los depuradores. Los iDevices Debes conocer y Para saber más con vídeos sobre el uso del depurador en Netbeans los pasaríamos a un nuevo apartado específico de depuración en Java con Netbeans

- Añadir un nuevo apartado 6.2.- Depurando código Java con Netbeans donde podemos incluir un breve resumen sobre las opciones más comunes del IDE Netbeans con capturas de pantalla para depurar código Java, que ya lo tenemos preparado de otros años como ayuda en las tareas. También se añadirían los iDevices que estaban en el apartado anterior.
Ubicación: Apartado 7.
Mejora (tipo 2): Reforzar el uso del depurador, detallando el proceso para depurar código con un vídeo o contenido interactivo.
Ubicación: varias preguntas
Mejora (Examen online): Eliminar las preguntas referentes a pruebas del software ya que dicho contenido no figura en la unidad
Ubicación: Depurador
Mejora (Mapa conceptual): Se ha ampliado también con un par de elementos más sobre la funcionalidad del depurador (ejecución paso a paso y visualización del contenido de variables o evaluación de expresiones).
Ubicación: rama pruebas
Mejora (Mapa conceptual): Eliminar la rama de pruebas de software que ya no está en el tema
Ubicación: Tabla de contenidos de la unidad.
Mejora (Orientaciones del alumnado): Se han incluido los cambios de nombres de apartados y apartados adicionales de la sección 6.

Versión: 02.00.00	Fecha de actualización: 11/06/21	Autoría: Diosdado Sánchez Hernández

Ubicación: Toda la unidad

Mejora (tipo 3): - Dedicar algunos apartados nuevos a trabajar con los operadores lógicos de manera que se vea cómo se puede incluir en una sola sentencia if lo que podría haber en varias.
- Incluir ejemplos donde se muestren alternativas de uso de if anidados o paralelos frente al uso de operadores lógicos en las condiciones para ver que un mismo conjunto de condiciones se puede expresar de muchas formas posibles.
- Dedicar algún apartado específico a contadores y otro a acumuladores, explicando su particular utilidad y su uso con abundantes ejemplos. Poner ejemplos de acumuladores que no sean solo sumadores, sino
también multiplicadores y cadenas que se van yuxtaponiendo.
- Incluir una sección o apartado dedicado a las variables locales o de bloque donde quede patente que en cada nuevo bloque Java pueden declararse.
- Eliminar la sección 4.2 y dejar su explicación en la unidad 4 donde es el primer lugar donde se puede utilizar. No tiene sentido poner un ejemplo de un array en esta unidad donde el alumno no sabe lo que es un array. Se podría hacer una referencia mínima en un Para saber más de la sección 4.1 o 4.1.1 donde se hiciera referencia a otros tipos de for que se verán más adelante cuando se estudien estructuras de datos tanto estáticas (arrays) como dinámicas (colecciones).
- Sección 5.1 - Incluir una sección 5.1.1 de ejemplo de lo que no se debe hacer con el break, con un ejemplo de bucle del que se sale con break indicando que eso es justo lo que no se debe hacer nunca, pues hay muchos alumnos que lo están haciendo.
- Transformar cada uno de los ejemplos resueltos del apartado 4.5 al formato iDevice Ejercicio resuelto donde se vaya desarrollando cada ejercicio mediante fragmentos de retroalimentación donde se vaya explicando paso a paso cómo resolver cada problema. Quitar los proyectos y dejar código limpio explicado y comentado para cada ejercicio específico, explicándolo detalladamente. Analizar los ejemplos que hay ahora y sustituir aquellos que no sean muy claros por otros que sean más fáciles de entender.
- Desarrollar el apartado de depuración con algo más de explicación del funcionamiento del depurador junto con algún vídeo explicativo. Incluir ejemplos de depuración de todas las estructuras de control:
- ejemplos de depuración de if,
- ejemplos de depuración de switch,
- ejemplos de depuración de for
- ejemplos de depuración de while
- ejemplos de depuración de do-while.

Ubicación: 3.1.- Estructura iva simple y iva compuesta: if / if-else.
Mejora (tipo 1): Ejercicio resuelto: en su última retroalimentación, en el código prouesto (en ambos ejemplos), en la sentencia if (numero 0) debería usarse = en lugar de para comparar si el número introducido es positivo o no, ya que de lo contrario el cero se consideraría positivo y no queremos que eso suceda. Por tanto, en ambos ejemplos de código debería incluirse: if (numero=0).
Ubicación: 3.2.1.- Estructura switch en Java. (Ejercicio Resuelto)
Mejora (tipo 1): Donde dice cálcule debe decir calcule
Ubicación: 3.2.- Estructura iva múltiple: switch.
Mejora (tipo 1): En el segundo párrafo donde dice seguiridad debe decir seguridad
Ubicación: Apartado 3 y 4.
Mejora (tipo 2): Dividir el apartado 4.5 (ejemplos resueltos), separando ejercicios que solo usen ifs y switchs, para ponerlos en un nuevo apartado dentro del apartado 3. Completar los ejemplos con casos más representativos.

Ubicación: Apartado 3

Mejora (tipo 2): Explicar como algunas condiciones de los ifs (y en ende, en los bucles) pueden reducirse:br>
if (a==true) --> if (a) br>
if (a==false) if (!a) br>
etc.
Ubicación: Apartado 4.
Mejora (tipo 2): añadir uno o más apartados que contengan explicaciones y ejemplos sobre estructuras anidadas (if dentro de if, if dentro de bucle, bucle dentro de bucle, etc.) con ejemplos.
Ubicación: 6.- Prueba de programas.
Mejora (tipo 2): Este apartado se elimina debido a que no entra dentro de los contenidos propuestos según normativa, ni tampoco dentro de los resultados de aprendizaje.
Ubicación: todo
Mejora (tipo 1): mas ejemplos en los temarios, paso por paso
Ubicación: Mapa conceptual
Mejora (Mapa conceptual): No se han añadido conceptos nuevos. Han sido cambios metodológicos y de inclusión de ejemplo. Permanece igual.
Ubicación: Todo el documento
Mejora (Orientaciones del alumnado): Adaptación de la lista de apartados conforme a las modificaciones realizadas en los contenidos. Eliminadas las referencias a pruebas y pseudocódigo. Añadidos algunos acrónimos que faltaban.

Versión: 01.04.00	Fecha de actualización: 09/10/20	Autoría: Diosdado Sánchez Hernández

Ubicación: Toda la unidad + 4.3.1 + 4.4.1 + 4.5
Mejora (tipo 2): Eliminar cualquier referencia al pseudocódigo en la unidad, suprimiendo las secciones 4.3.1 y 4.4.1 completamente, así como eliminando de la sección 4.5 todos los ejemplos resueltos en pseudocódigo.
Ubicación: 5.3.- Sentencia return
Mejora (tipo 2): Eliminar completamente la sección 5.3 sobre la sentencia return.
Se trata de una instrucción que no podrá ser utilizada hasta que el alumnado comience a implementar métodos en sus propias clases allá por la unidad 5. Es en esa unidad es donde se debe explicar y se deben incluir ejemplos apropiados para la sentencia return.
Incluir aquí una explicación de esta sentencia no tiene ninguna utilidad práctica y tan solo puede llevar a confundir al alumnado, pues no le va a resultar de utilidad alguna.
Ubicación: 5.3.- Sentencia return.
Mejora (tipo 1): En la sección de autoevaluación, la segunda opción no queda claro por qué es incorrecta. Estaría bien aclarar con un ejemplo que una sentencia return vacía solo está permitida en métodos void. En los métodos en los que se devuelve un tipo u objeto no está permitida.
Ubicación: Apartado 4.5. Ejercicios resueltos.
Mejora (tipo 1): El ejercicio 8 muestra NaN cuando el número de números positivos introducidos son 0, y cuando el número de números negativos introducidos son 0.
Además, lo ejercicios de este apartado se abren en la misma página, y no en otra página _blank.
Ubicación: Introducción
Mejora (tipo 1): Se ha eliminado el la primera frase del penúltimo párrafo que decía: Además, como nuestros programas podrán generar errores y situaciones especiales, echaremos un vistazo al manejo de excepciones en Java ya que esta sección ha cambiado al tema siguiente.
Ubicación: Introducción
Mejora (tipo 1): Eliminar el la primera frase del penultimo parrafo que dice: Además, como nuestros programas podrán generar errores y situaciones especiales, echaremos un vistazo al manejo de excepciones en Java ya que esta sección ha cambiado al tema siguiente.
Ubicación: Mapa conceptual
Mejora (Mapa conceptual): Adaptación del mapa conceptual a los nuevos contenidos (desaparece la introducción a la sentencia return, dejándose para más adelante).
Ubicación: Índice de la unidad
Mejora (Orientaciones del alumnado): Adaptación del índice en la guía al índice real en la unidad (había secciones que no coincidían).

Versión: 01.03.00	Fecha de actualización: 01/10/19	Autoría: Salvador Ror

Ubicación: 6.- Prueba de programas.
Mejora (tipo 2): Se propone mover este apartado a un anexo (prueba de programas) debido a dentro de los contenidos propuestos según normativa, ni tampoco dentro de los resultados de considera que es material complementario.
Ubicación: Apartado 8.- Documentación del código.
Mejora (tipo 2): Se propone trasladar este apartado a la unidad 3 (sería el nuevo apartado 9), alumnado se centre en aprender las estructuras de control, que son extremadamente relevante

Ubicación: 4.5
Mejora (tipo 1): El ejemplo resuelto número 9 del apartado 4.5, pone que se usan do-while y if enunciado no da pie a ello. En pseudocódigo de hecho no se usa, aunque en Java si, obligand horas sean mayores o iguales a 0.
La idea es cambiar el ejercicio y el código en pseudocódigo para que se emplee dicha estructu
Ubicación: 4.5
Mejora (tipo 1): El ejercicio 2 tiene un pequeño fallo lógico. Si el primer número que le introdu primeras es un número negativo, dice que el resultado de la multiplicación es 1, pero no ha lleg multiplicar nada. No coincide con el pseudocódigo aportado.
Ubicación: Apartado 7
Mejora (tipo 1): Primer enlace de la página roto, sobre depuración en el debes conocer. Debe

https://web.archive.org/web/20120709000731/http://losremediosinformaticos.blogspot.com/200 de-programas-con-netbeans.html

Ubicación: Apartado 4.5
Mejora (tipo 1): En el enunciado del ejercicio 7 del apartado 4.5 pone:

Hacer un programa que intercambie el valor de dos variable, a y b, cuyos valores se han leído
El tipo de las variables, puede ser cualquiera en realidad, pero va a suponer que es entero.

El programa en Java hace justamente eso, pero el pseudocódigo hace el intercambio solo si A
Por lo que está un poco todo liado. El enunciado debería ser, algo como sigue:

Hacer un programa que intercambie el valor de dos variable, a y b, cuyos valores se han leído solo si el valor de a es mayor al valor de b.
El tipo de las variables, puede ser cualquiera en realidad, pero va a suponer que es entero.

Y la solución en Java sería algo así:

```
public class JavaEjercicio7 {

 **/
* Este programa todavía presupone que el usuario sólo va a introducir por
* teclado números enteros válidos. Todavía no se controlan los errores de
* entrada, ni se capturan excepciones.
*
* Se trata de un programa que lee el valor de dos variables desde teclado,
* muestra el valor que contiene cada variables, intercambia sus valores
* usando una variable auxiliar si el primero es mayor que el segundo,
* y vuelve a mostrar sus valores a modo de comprobación.
*
* @param args de argumentos para la línea de comandos.
*/
public static void main(String[] args) {
Scanner teclado= new Scanner(System.in);
int a, b, auxiliar;
 *     /
* Asignamos valores para a y b, leyéndolos desde teclado.
*/
System.out.println(Introduce un valor entero para la variable a:);
a=teclado.nextInt();
System.out.println(Introduce un valor entero para la variable b:);
b=teclado.nextInt();
```

```
System.out.println(Inicialmente, los valores de las variables son; a= +a+ b= +b+.);
 *    /
* intercambiamos los valores
*/
if (a > b) {
auxiliar = a;
a = b;
b = auxiliar;
       }

 *    /
* Mostramos los valores finales de las variables
*/
System.out.println(Tras el intercambio, los valores de las variables son; a= +a+ b= +b+.);
   }
}
```

Ubicación: Mapa conceptual
Mejora (Mapa conceptual): Se cambia el mapa conceptual para que no aparezca la parte de que ahora pasa a estar en la unidad 3.
Ubicación: Orientaciones para el alumnado
Mejora (Orientaciones del alumnado): Se cambia la redacción del apartado introductorio par mejor con los cambios de tipo 2 realizados en la unidad (se traslada la documentación a la uni mueve el apartado de prueba de programas a un anexo).
Ubicación: Orientaciones para el alumnado
Mejora (Orientaciones del alumnado): Dice: Esta tercera unidad, cuando debería decir: En e unidad te centrarás

Versión: 01.02.00	Fecha de actualización: 25/09/18	Autoría: Salvador Romero Villegas

Ubicación: Varios Apartados
Mejora (tipo 2): - El ejemplo del apartado 6.3 usa un método de instancia, y ese momento no se han visto los métodos. En vez de quitar el ejemplo, se propone mover el apartado 6 a la unidad 3, justo después del apartado 7 (entrada y salida), donde ya existe una noción básica de métodos. Además, el ejercicio resuelto del apartado 6.1. usa BufferedReader, algo que no se ve hasta la unidad 7, y haciendo este cambio se podría sustituir por algo relacionado con la clase Scanner. - Apartado 3.1.- El Ejercicio resuelto creo que sería buena idea plantearlo como un ejercicio real, es decir un enunciado que invite al alumnado a hacerlo. - Apartado 3.2.2.- El ejercicio resuelto creo que sería buena idea replantearlo como un ejercicio resuelto. En vez de poner Acceder al ... poner Crea un programa en Java que haga lo siguiente.... - Apartado 4.1.2 hay un debes conocer que creo que debería plantearse como ejercicio resuelto (haz un programa en java que calcule la tabla de un número leído por teclado). - Apartado 4.3.2 hay también un debes conocer que nuevamente pienso que debería plantearse como un ejercicio resuelto. - Apartado 4.4.2 hay otro debes conocer que pienso que debería también plantearse como un ejercicio resuelto. - Anexo I es conveniente que se traslade al apartado 4.5.- Ejercicios. Al estar como anexo parece que se le quita la importancia que tienen estos ejercicios y habría que hacer hincapié en la necesidad de que practiquen estos enunciados. - Aunque no esté recomendado el uso de break y continue, no estaría mal poner un ejercicio resuelto en el apartado 5.1.
Ubicación: Varios.
Mejora (tipo 1): - El ejemplo del apartado 6.3 usa un método de instancia, y ese momento no se han visto los métodos. En vez de quitar el ejemplo, se propone mover el apartado 6 a la unidad 3, justo después del apartado 7 (entrada y salida), donde ya existe una noción básica de métodos. Además, el ejercicio resuelto del apartado 6.1. usa BufferedReader, algo que no se ve hasta la unidad 7, y haciendo este cambio se podría sustituir por algo relacionado con la clase Scanner.

- Apartado 3.1.- El Ejercicio resuelto creo que sería buena idea plantearlo como un ejercicio real, es decir un enunciado que invite al alumnado a hacerlo.

- Apartado 3.2.2.- El ejercicio resuelto creo que sería buena idea replantearlo como un ejercicio resuelto. En vez de poner Acceder al ... poner Crea un programa en Java que haga lo siguiente....

- Apartado 4.1.2 hay un debes conocer que creo que debería plantearse como ejercicio resuelto (haz un programa en java que calcule la tabla de un número leído por teclado).

- Apartado 4.3.2 hay también un debes conocer que nuevamente pienso que debería plantearse como un ejercicio resuelto.

- Apartado 4.4.2 hay otro debes conocer que pienso que debería también plantearse como un ejercicio resuelto.

- Anexo I es conveniente que se traslade al apartado 4.5.- Ejercicios. Al estar como anexo parece que se le quita la importancia que tienen estos ejercicios y habría que hacer hincapié en la necesidad de que practiquen estos enunciados.

- Aunque no esté recomendado el uso de break y continue, no estaría mal poner un ejercicio resuelto en el apartado 5.1.
Ubicación: No especificada.
Mejora (tipo 1): Eliminación del idevice de licencia.

Versión: 01.01.00	Fecha de actualización: 30/10/13	Autoría: José Javier Bermúdez Hernández
Se han introducido las estructuras de control primero en pseudocódigo, y luego en Java; se ha explicado mejor los casos en los que resulta necesario en Java usar estructuras de salto incondicional; se ha introducido un apartado de documentación externa que aparece en normativa y no estaba recogido en contenidos; se ha incluido un anexo de ejemplos resueltos en pseudocódigo y en Java. Se ha actualizado el mapa conceptual, y las orientaciones del alumnado.		

Versión: 01.00.00	Fecha de actualización: 30/10/13
Versión inicial de los materiales.	