Programación

Bloque 02 - Estructuras de control

Índice

1.- Introducción

Hasta ahora todos los programas realizados eran necesariamente simples por una razón también simple: Sólo podíamos usar instrucciones y expresiones en forma *secuencial*, esto es, una instrucción siguiendo a otra y *todas* las instrucciones debían hacerse siempre.

Esta forma de funcionar, aunque sirve para hacer algunas tareas de programación no sirve para hacerlas todas ya que muchas veces es necesario el poder:

- Hacer un paso o pasos en un programa sólo algunas veces, dependiendo de los valores de ciertos datos.
- Repetir un mismo paso o grupo de pasos más de una vez sin necesidad de conocer a priori cuantas repeticiones se van a realizar (también depende en cierta medida de los datos).

Es pot esto que todos los lenguajes de programación necesitan ciertas **estructuras de control** cuya misión es reconfigurar el flujo o secuencia de ejecución de un programa según los datos del mismo, de forma que se puedan realizar cualquier tarea, por compleja que esta sea.

En este bloque aprenderemos cuales estructuras tenemos a nuestra disposición asi como la forma de usarlas en Java.

2.- Estructuras de selección

Las estructuras de selección son aquellas que permiten **seleccionar** (de ahi su nombre) qué instrucciones o pasos se realizan en un momento determinado de ejecución del programa. También se conocen comunmente como estructuras condicionales, aunque el nombre correcto y formal es el arriba indicado.

Hay dos tipos de estructuras de selección clásicas: La estructura simple (que a veces se puede complicar bastante) y la estructura multiple.

2.1.- Estructura if

La primera estructura de selección simple es el implementado por la instrucción `if` de la forma:

```
if (valor_booleano) {
  .....instrucciones....
}
```

La instrucción consta de la palabra reservada `if`, seguida de un valor booleano entre paréntesis (los paréntesis son obligatorios), y seguido de un bloque.

El funcionamiento es muy simple: si el valor booleano vale `true`, se ejecutarán las instrucciones contenidas en el bloque. Si vale `false`, las instrucciones contenidas en el bloque se ignoran y continúa la ejecución por la instrucción situada **detrás** del bloque.

Por supuesto, para proporcionar el valor booleano se puede emplear cualquier expresión que devuelva un valor booleano, desde un literal (true, false) a una expresión lógica, de comparación o el valor devuelto por un mensaje. Si la expresión no devuelve un valor booleano, Java lo considera un error y no compila.

Ejemplo:

```
System.out.print("Introduzca un número entero: ");
int numero = Integer.parseInt(sc.nextLine());
if (numero % 2 == 0) {
  System.out.println("El número " + numero + " es par");
}
System.out.println("Fin del programa");
```

Este programa solicita un número entero al usuario desde teclado.

A continuación si el número es par (`numero % 2 == 0` vale `true` si el número es par y `false` si no lo es) se ejecuta el contenido del bloque, el cual imprime por pantalla el texto `El número ... es par`. Si el número no es par, no es imprime nada. En cualquier caso **siempre** se imprime el texto de la última línea `Fin del programa`.

Como puedes ver la instrucción de la línea 4 se ejecutará unas veces si y otras no, dependiendo del valor que introduzca el usuario desde teclado. Esta es la función que realiza la sentencia `if`.

2.2.- Estructura if-else

El ejemplo del apartado anterior está algo "cojo". Esto es asi porque si el número introducido es par se muestra un mensaje acorde pero si no lo es no se imprime nada. Lo ideal sería que cuando el número sea impar se imprima un mensaje indicándolo. Podemos ampliar el programa del ejemplo para que haga esto:

```
System.out.print("Introduzca un número entero: ");
int numero = Integer.parseInt(sc.nextLine());
if (numero % 2 == 0) {
  System.out.println("El número " + numero + " es par");
}
if (numero %2 != 0) {
  System.out.println("El número " + numero + " es impar");
}
System.out.println("Fin del programa");
```

Esta versión hace lo que hemos dicho. Lo nuevo es lo introducido en las líneas 6 a 8. Se trata de otra sentencia `if`, que emplea una expresión booleana diseñada para ser la completamente opuesta a la usada en la línea 3, de forma que cuando ésta valga `true`, la de la línea 6 valga `false` y viceversa. Si examinas cuidadosamente el código podrás comprobar que si en una ejecución del programa se ejecuta la línea 4 no se ejecuará la 7 y viceversa.

Esta forma de usar la estructura `if` es tan común que se ha inventado una forma de expresarla de forma más sencilla: usando la sentencia `else`. Ésta tiene la forma:

```
if (valor_booleano) {
  ..instrucciones bloque if...
} else {
  ..instrucciones bloque else...
}
```

En esta forma se añade, después del bloque correspondiente al `if` la palabra reservada `else` y otro bloque de instrucciones. El funcionamiento es el siguiente: Si el valor booleano es `true` se ejecutarán las instrucciones contenidas en el bloque después de `if`. Las instrucciones contenidas en el bloque correspondiente a `else` se ignorarán y no se ejecutarán. En caso que el valor booleano fuera `false` ocurriría lo opuesto: las instrucciones contenidas en el bloque `if` **no** se ejecutarían y si lo harían las contenidas en el bloque `else`.

Esta forma tienes dos ventajas sobre la anterior. Una es obvia (es más corto) y otra no lo es tanto: En esta segunda forma no es necesario el diseñar dos expresiones booleanas (que en algunos casos pueden llegar a ser bastante complejas de por si) exactamente opuestas. La sentencia `else` se asegura automáticamente que se realizarán instrucciones siempre que ocurra la condición opuesta a la expresada en el `if`, sea esta lo compleja que sea. Por lo tanto es más sencillo de programar y menos propenso a errores.

2.3.- Estructura if - (else if) - else

Existe otro caso menos común que los dos anteriores pero que se da con cierta frecuencia, que es el de elegir entre más de un caso. Por ejemplo, supongamos que queremos hacer una tarea distinta

sobre una persona dependiendo de si es menor de edad, adulto "normal" o adulto "jubilado". En el primer caso la edad de la persona debe ser menor de 18 años, en el segundo debe estar entre 18 y 65, ambos incluidos y en el tercero debe ser mayor de 65.

Aqui no tenemos una elección entre dos opciones, como era el caso del if ... else sino que tenemos tres. Para ello existe una forma especial de if que nos permite realizarla de forma más o menos simple.

En el caso de nuestro ejemplo, esta estructura sería:

```
if (persona.edad < 18) {
  ..... hacer el tratamiento que sea para menores de edad...
} else if ((persona.edad >= 18) && (persona.edad < 65)) {
  ..... hacer el tratamiento para adultos no jubilados .....
} else {
  ...... hacer el tratamiento para adultos jubilados ....
}
.... siguientes ....
```

El funcionamiento de esta esructura es el siguiente:

- Si el valor booleano del primer `if` es `true` se ejecuta el bloque que le sigue.
- Si no lo es, se examina el valor correspondiente al segundo `if`. Si es `true` se ejecuta el bloque que lo sigue.
- Si tampoco el segundo es `true`, se ejecuta el bloque correspondiente al `else`.

En los tres casos, una vez finalizado el código del bloque correspondiente, se continúa por el código marcado como `siguientes`.

2.4.- Operador ternario

Un uso muy común de la instrucción if / else es el de asignar un valor determinado a una variable dependiendo de una condición. Por ejemplo:

```
System.out.print("Introduce un número entero: ");
int numero = Integer.parseInt(sc.nextLine());
String parImpar;
if (numero % 2 == 0) {
  parImpar = "par";
} else {
  parImpar = "impar";
}
System.out.println("El número introducido es " + parImpar);
```

Si sigues el código verás que en las líneas 4 a 8 lo que se hace es asignar un valor u otro (par o impar) a la variable `parImpar`, dependiendo de una condición (en este caso `numero % 2 == 0`).

Este tipo de uso, como ya se ha dicho, es muy común, por lo que muchos lenguajes, entre ellos Java, ofrece una forma abreviada de hacerlo: el operador ternario.

El operador ternario se llama asi porque recibe tres operandos en lugar de dos o uno como viene siendo habitual. Tiene la forma:

```
valor_booleano ? valor_si_true : valor_si_false
```

Como ves, toma tres valores, de los cuales el primero debe ser obligatoriamente booleano. El operador devuelve el valor valor_si_true si el valor booleano es true o el valor valor_si_false si el valor booleano es false. De esta forma, el ejemplo anterior podría quedar, de forma más simple como sigue:

```
System.out.print("Introduce un número entero: ");
int numero = Integer.parseInt(sc.nextLine());
String parImpar = (numero % 2 == 0) ? "par" : "impar";
System.out.println("El número introducido es " + parImpar);
```

La línea 3 asigna el valor `"par"` o `"impar"` a la variable `parImpar` dependiendo del valor booleano que produzca la expresión `numero % 2 == 0`. Nótese que esta expresión, aunque no debe ir obligatoriamente entre paréntesis, se suele incluir entre ellos como costumbre normalmente aceptada que se recomienda seguir.

2.5.- Estructura de selección múltiple (switch)

Con las estructuras vistas hasta ahora disponemos de las herramientas necesarias para ejecutar o no a nuestra conveniencia cualquier instrucción o bloque de instrucciones.

Sin embargo, la estructura de selección múltiple que hemos visto hasta ahora (`if - else if - else if... else`) es compleja y difícil de seguir cuando hay más de tres o cuatro casos ya que hay que ir buscando condiciones y bloques y se nos puede trastocar alguno si no leemos con atención.

Para simplificar este tipo de estructuras se inventó otra forma de hacer la selección múltiple que es más clara de ver en muchos casos: La estructura `switch`.

La estructura es como sigue:

```
switch (valor_primitivo) {
  case literal_o_constante1:
    instruccion_bloque_1;
    instruccion_bloque_1;
  case literal_o_constante2:
    instruccion_bloque_2;
    instruccion_bloque_2;
    break;
  default:
    instruccion_bloque_3;
```

```
    instruccion_bloque_3;
}
```

Lo que hace la sentencia es comparar el valor `valor_primitivo` (que debe ser `int`, `char` o `String`) con cada uno de los valores que se proporcionan en las distintas clausulas `case`, en el orden en que estas aparecen en el código. En el momento en que se produce una coincidencia entre los dos valores, se comienzan a ejecutar las sentencias que siguen al `case` en cuestión, hasta el final del `switch`. Nótese los dos puntos (`:`) colocados después de los valores de cada `case` (son obligatorios). Si no se produce ninguna coincidencia y existe un apartado `default`, se ejecuta lo que haya tras éste. Si no se produce ninguna coincidencia y NO existe un apartado `default`, no se haría nada. Hay que insistir en el hecho de que `default` no lleva valor alguno y es una palabra reservada.

Este comportamiento tiene unas consecuencias un poco inesperadas que deben ser tratadas con cuidado. Sin embargo, bien utilizado este comportamiento también puede ser útil.

Examinemos el siguiente código:

```
System.out.print("Seleccione la operación a realizar (s: suma, r: resta): ");
char opcion = sc.nextLine().charAt(0);
int resultado;
switch (opcion) {
  case 's':
    resultado = numero1 + numero2;
  case 'r':
    resultado = numero1 - numero2;
}
System.out.println("El resultado de la operación es: " + resultado);
```

En las dos primeras líneas se solicita al usuario un carácter que indique qué operación se quiere realizar (`'s'` para la suma, `'r'` para la resta).

A continuación se declara la variable que va a almacenar el resultado y comienza una sentencia `switch`, usando como `valor_primitivo` el contenido en la variable `opcion`. Si este vale `'r'` se ejecutará la sentencia contenida en la línea 8, se acabará el `switch` y se imprimirá el resultado en la línea 10, que será correcto. Sin embargo, ¿qué ocurre si el usuario introduce el carácter `'s'`?. Pues en este caso, en el `switch` se comparan los valores con `'s'`. Se elige por tanto el primer `case` (linea 5) y se ejecuta la operación de la línea 6. Hasta aqui todo bien. Lo que puede resultar insólito a simple vista es que a continuación **ejecuta la sentencia contenida en la línea 8**, con lo que el resultado será también la resta, cosa que obviamente no queríamos.

¿Por qué ocurre esto? Esto ocurre porque hemos dicho que `switch` compara el valor con cada case hasta que encuentra la primera coincidencia y ejecuta desde ahi **hasta el final del switch**. Esto incluye las instrucciones contenidas en los `case` situados bajo el elegido.

Este comportamiento, que obviamente no es el deseado en muchos casos, se puede modificar usando una nueva instrucción, `break`, que colocada como última instrucción dentro de un `case` provoca que se termine el `switch` inmediatamente al llegar a ella y se continúe por las instrucciones situadas a continuación del mismo. El código anterior funcionaría bien haciendo:

```
System.out.print("Seleccione la operación a realizar (s: suma, r: resta): ");
char opcion = sc.nextLine().charAt(0);
int resultado;
```

```
switch (opcion) {
  case 's':
    resultado = numero1 + numero2;
    break;
  case 'r':
    resultado = numero1 - numero2;
    break;
}
System.out.println("El resultado de la operación es: " + resultado);
```

Se han añadido dos sentencias `break` en las líneas 7 y 10. La de la línea 10 no es estrictamente necesaria ya que no hacer realmente nada (de todas formas se saldría del `switch` dado que es la última instrucción) pero se suele escribir también por consistencia.

Un ejemplo de cuando es útil el comportamiento "en cascada" de `switch` sería el siguiente:

```
System.out.print("Introduce una letra: ");
char letra = sc.nextLine().charAt(0);
switch (opcion) {
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
    System.out.println("La letra es una vocal");
    break;
  case 'y':
    System.out.println("La letra a veces es una vocal y a veces consonante");
    break;
  default:
    System.out.println("La letra es una consonante");
    break;
}
```

En este caso, el usuario introduce una letra desde teclado (lineas 1 y 2). En el switch se usa el carácter leido para elegir un `case`. Dado que cuando se encuentra una coincidencia se ejecuta hasta el final (o hasta que se encuentre un `break`), en caso de que el valor sea `'a'` se continúa por la 'e', después por la `'i'`, etc. hasta que se llega a la `'u'` en la cual se imprime el mensaje y se termina (`break`). Lo mismo ocurre con la `'e'`, `'i'`, `'o'` e `'u'`. En el caso de la `'y'` se considera una medio vocal, medio consonante y si no es ninguna de las anteriores entra por `default`. Aqui se imprime que es una consonante ya que es el caso que queda.

3.- Ámbito de las variables

Hasta ahora no hemos tratado el problema del ámbito de las variables debido a que no habiamos usado bloques anidados (bloques dentro de bloques). En esta unidad, sin embargo, ya comenzamos a usar estas estructuras por lo que el problema se presenta y debemos saber como solucionarlo.

Hasta ahora todas las variables que hemos creado eran válidas desde el momento en que se creaban hasta el final del bloque. Esto era asi porque sólo había un bloque en todo el programa (el bloque del método `main`). Al introducir las estructuras de control nos ocurre que ahora nuestros programas contienen varios bloques anidados y estas reglas de existencia de las variables no funcionan ya exactamente igual, o asi nos lo parece.

La regla anterior habría que modificarla y expresarla de manera ligeramente distinta. La modificación es que la regla quedaría asi:

Una variable existe desde el momento en que se crea hasta el momento en que termina el bloque en donde se declara.

Por ejemplo, si intentas escribir el siguiente código:

```
int a = 2;
if (a > 0) {
  int i = 5;
  System.out.println(a);
}
System.out.println(i);
```

Descubrirás que no funciona. ¿A qué se debe esto? Se debe a que en la línea 5 se está intentando utilizar la variable `i`, ¡que ya no existe!. ¿A qué se debe esto? A la aplicación de la regla que acabamos de enunciar: Una variable sólo existe desde el momento en que se crea hasta el final del bloque que la contiene. Como se crea dentro del bloque del `if`, deja de existir al acabar éste por lo que en la línea 5 ya no existe y, por tanto, no se puede usar. Es como si nunca se hubiera declarado. En cambio, el siguiente código si funcionaría:

```
int a = 2;
int i = 0;
if (a > 0) {
  i = 5;
  System.out.println(a);
}
System.out.println(i);
```

Ahora `i` si existe porque se declaró en el mismo bloque en que se usa y este aún no ha terminado.

La variable a, sin embargo puede usarse sin problemas en los dos bloque. Esto es asi porque se declara en el bloque exterior y este aún no ha finalizado cuando se entra al bloque interior, por lo que se puede usar en éste sin problemas.

Hay que tener cuidado con esta regla ya que nos podemos encontrar con problemas inesperados, por ejemplo, declarar una variable dentro de un bloque `switch` que luego nos encontramos que no podemos usar al terminar el mismo, por ejemplo.

4.- Estructuras de repetición

Una situación que se produce frecuentemente a la hora de realizar algoritmos es la necesidad de repetir varias veces una serie de instrucciones, con ligeras diferencias entre una repetición y otra. Por ejemplo, si se quiere imprimir una lista de los números del 1 al 100, la instrucción es la misma (imprimir), se debe repetir 100 veces y debe cambiar ligeramente de una repetición a otra (el número a imprimir en cada repetición debe ser distinto).

A las estructuras que sirven para implementar estas repeticiones se les denomina de forma apropiada estructuras de repetición, también conocidas como ciclos o bucles.

4.1.1.- Estructura while

La estructura `while` repite un bloque de instrucciones mientras (de ahi su nombre) una condición sea verdadera. Lo que distingue a while de otras estructuras de repetición es que la comprobación de la condición se realiza **antes** de iniciar una repetición y puede incluso ocurrir que el bloque no se ejecute nunca si la primera vez que se comprueba la condición, ésta es falsa.

La estructura `while` tiene la forma:

```
while (expresión_booleana) {
  ..... instrucciones que se repiten....
}
```

Consta de la palabra reservada `while`, seguida de una expresión booleana entre paréntesis (obligatorios). Por último un bloque de instrucciones.

Funciona de la manera siguiente: Se evalúa la expresión booleana. Si es `true` se ejecuta el contenido del bloque. Al finalizar la ejecución del bloque se vuelve a comprobar la expresión booleana y vuelta a empezar. Si la expresión vale `false`, se salta el bloque y se continúa por la siguiente instrucción.

Ejemplo:

```
FuenteDeNumerosEnterosAleatorios fuente = new
FuenteDeNumerosEnterosAleatorios(1, 100);
while (fuente.hayMasNumeros()) {
  int numero = fuente.siguienteNumero();
  System.out.println("El numero es " + numero);
}
```

Este código crea un objeto de la clase (ficticia) `FuenteDeNumerosAleatorios` que tiene dos mensajes: `hayMasNumeros` que devuelve `true` si quedan más números en la fuente o `false` si no y `siguienteNumero` que devuelve el siguiente número en la fuente. A continuación se hace un ciclo que va leyendo números de la fuente mientras haya en ésta y los va imprimiendo por pantalla. Aqui hay que usar un bloque while porque puede que la fuente esté vacía en un principio y no se logre leer ningún número.

4.1.2.- Estructura do-while

La estructura do-while es muy similar a la anterior pero tiene una diferencia fundamenta. Tiene la forma:

```
do {
  ..... instrucciones....
} while (expresion_booleana);
```

Como se puede ver comienza por la palabra reservada do, seguida de un bloque con las instrucciones a repetir, seguido todo por la palabra reservada while y una expresión booleana entre paréntesis (obligatorios).

En este caso lo primero que se hace es ejecutar el bloque de instrucciones y a continuación se evalúe la expresión booleana. Si vale true se vuelve a ejecutar el bloque otra vez. Si no lo vale se termina y continúa por la siguiente instrucción.

Nótese que la diferencia fundamental con while es que en este caso el cuerpo del ciclo (el bloque de instrucciones) se ejecuta **siempre** al menos una vez, a diferencia de while en que podía ocurrir el caso de que el cuerpo no se ejecutara ninguna vez.

Ejemplo:

```
do {
  System.out.print("Introduce un número entero (0 para terminar): ");
  int numero = Integer.parseInt(scan.nextLine());
  if (numero != 0) {
    System.out.println("El numero " + numero + " es " + ((numero % 2 == 0) ?
"par" : "impar"));
  }
} while (numero != 0);
```

En este caso se van pidiendo números por teclado hasta que se introduce un cero. Por cada número introducido (exceptuando el cero) se imprime por pantalla si el número es par o impar.

Nótese que siempre hay que pedir un número por lo menos una vez (aunque el primer número que se introduzca sea el cero), por lo que debemos usar do-while.

4.1.3.- Estructura for

La estructura `for` es la más complicada de las tres estructuras de repetición y tiene más de una forma de funcionamiento. En este bloque veremos la más simple pero la revisitaremos y ampliaremos cuando hablemos de los arrays y las colecciones para ver una nueva forma de `for`.

La estructura `for` tiene la forma:

```
for
(expresion_inicializacion;expresion_continuar;expresion_actualizac
ion) {
  .... instrucciones...
}
```

Funciona de la siguiente manera:

1. Se evalúa la expresión `expresion_inicializacion`. Esta expresión sólo se evalúa una vez y sirve para realizar inicializaciones de cosas que se van a usar en el bucle, como variables, normalmente
2. Se evalúa la expresión booleana `expresion_continuar`. Si vale `true` se ejecuta el bloque. Si vale `false` se salta el bloque y se continúa por la siguiente instrucción
3. Después de ejecutar el bloque se evalúa la expresión `expresion_actualizacion`.
4. Se vuelve al paso 2

Tanto los paréntesis como los punto y coma (`;`) son obligatorios. Las expresiones pueden no incluirse si no son necesarias pero los punto y coma deben estar siempre (si no se produce un error de compilación). Si la expresión `expresion_continuar` no se indica se supone que vale siempre `true`.

Ejemplo:

```
// Imprime los números del 1 al 100
for (int i = 0; i <=100; i++) {
  System.out.println(i);
}
```

Imprime los números enteros desde 1 al 100, ambos incluidos.

4.1.4.- Consideraciones a la hora de escribir repeticiones

Como hemos podido ver existen varias estructuras que se pueden usar para realizar repeticiones.

En primer lugar hay que indicar que teóricamente sólo sería necesaria la estructura while, pudiéndose simular las otras dos con ésta. Se deja como ejercicio el ver cómo esto es posible.

Sin embargo la disponibilidad de varias estructuras de repetición obedece a una razón de comodidad y eficiencia. Es más rápido, sencillo y eficiente utilizar la estructura adecuada para cada ocasión de forma que se minimicen errores y el código sea el más claro posible.

Por lo tanto es importante el saber elegir cual es la estructura más adecuada para cada situación que se nos pueda plantear mientras realizamos un programa.

Aunque no existe una única regla al respecto, una que puedes usar y que suele funcionar bien en la mayoría de las ocasiones es la siguiente:

En primer lugar, debes hacer y contestar a la siguiente pregunta: ¿Cuando el flujo del programa llega al inicio del ciclo, se tiene ya información sobre cuantas repeticiones se van a hacer en éste?. Dicho de otra manera, ¿el programa dispone de suficiente información al inicio del ciclo para saber cuantas repeticiones serán? Si la respuesta es *SI* la mejor opción es usar la estructura `for`.

Si la respuesta a la pregunta anterior es *NO* deberíamos hacernos otra a fin de elegir entre las otras dos opciones que nos quedan. La pregunta sería: ¿Es necesario que el cuerpo del ciclo (el bloque que se repite) se ejecute al menos una vez **siempre**? Si la respuesta es *SI* la mejor opción es usar `do-while`. Si la respuesta es *NO* entonces `while`.

Otra precaución al trabajar con ciclos es la de evitar lo que se conoce como ciclos infinitos. Estos son ciclos que nunca terminan de repetirse, principalmente por un mal diseño de la condición de salida del mismo. Cuando se entra a un ciclo infinito el programa se detiene (se dice que se cuelga) aunque realmente no para de funcionar ejecutando la misma sección de programa una y otra vez.

Para evitar ciclos infinitos debes asegurarte que:

- La condición de finalización del ciclo contiene al menos una variable o envío de mensaje que devuelve un valor. Si la expresión contiene sólo elementos constantes no cambiará nunca provocando o que el ciclo sólo se haga una (o ninguna) veces o bien que se produzca un bucle infinito. La condición siempre debe depender de algún valor que pueda variar.
- Dentro del cuerpo del ciclo se modifica eventualmente el valor de la variable o variables que participan en la condición de forma que eventualmente se alcance el final de las repeticiones. En el caso de mensajes a objetos esto queda fuera del alcance del programador que realiza el ciclo.

5.- Sentencias de salto

Aunque no es muy recomendable usarlas y en la mayoría de los casos se puede pasar sin ellas, Java también incluye sentencias de salto que pueden ser empleadas para simplificar la lógica, especialmente en ciclos.

5.1.- Sentencia break

La sentencia `break`, cuando se ejecuta, finaliza inmediatamente la ejecución del bloque en que se encuentra y continúa la ejecución por las instrucciones situadas inmediatamente a continuación del mismo.

5.2.- Sentencia continue

La sentencia `continue`, cuando se ejecuta, finaliza inmediatamente la ejecución del resto de instrucciones que contiene el bloque en que se encuentra. Si el bloque forma parte de un ciclo, la condición de éste se reevalúa como si se hubiera llegado al final del bloque de la forma habitual. Lo podemos ver como un reinicio de la repetición del ciclo para el siguiente valor.

6.- Control de excepciones

El control de excepciones en Java permite tratar las posibles condiciones excepcionales o erróneas que se produzcan y continuar la ejecución de un programa.

6.1.- Excepciones

En Java las condiciones excepcionales o excepciones son condiciones que se producen de forma excepcional durante la ejecución de un programa (no confundir con errores de sintaxis o similares que se detectan durante la compilación) y provocan que el programa no pueda continuar haciendo lo que fuera que estaba haciendo hasta ese momento.

Por ejemplo, supongamos que tenemos un programa que solicita dos números al usuario, divide al primero entre el segundo y muestra el resultado. ¿Qué ocurre si el usuario, maliciosamente o por error, introduce cero como segundo número? Como sabrás la división por cero es una operación que no se puede realizar ya que su resultado es indeterminado. ¿Qué ocurre cuando el programa se encuentra con que se pide dividir un número por cero? No puede obtener un resultado correcto (porque no lo hay) pero el código siguiente espera que el resultado se haya realizado. Como puedes ver se produce una condición excepcional, una condición que no está prevista y en la cual el programa no puede continuar.

Otro ejemplo es cuando solicitamos al usuario un número por teclado usando Integer.parseInt(scan.nextLine()) ¿Qué ocurre si el usuario, por error, teclea algo que no es un número? No se puede obtener un número correcto, porque no hay equivalencia pero no se puede seguir porque no tenemos el número esperado. Tenemos otra situación excepcional.

Lo que hace Java es lo que se conoce como *lanzar una excepción*. Se crea un objeto especial (la excepción) que contiene información sobre lo ocurrido (qué ha pasado y en qué instrucción exacta ha ocurrido) y se interrumpe inmediatamente la ejecución en la instrucción que provoca el error.

Si esta excepción no se trata de forma adecuada, el programa se termina inmediatamente, mostrando la información de la excepción por pantalla

6.2.- Proceso de excepciones

Si un programador no desea que un programa se interrumpa cuando se produzca una excepción determinada puede crear lo que se denomina un **manejador de excepciones**. Un manejador de excepciones en un bloque de código que se ejecuta de forma automática cuando se produce una excepción. El código dentro del bloque deberá tomar las medidas que crea oportunas para contener el daño producido por la excepción e incluso retomar la ejecución de la aplicación.

Un manejador de excepciones tiene la forma:

```
try {
  .... bloque de código normal.....
} catch (Excepcion1 e) {
  ... codigo del manejador de excepciones 1...
} catch (Excepcion2 e2) {
```

```
  .... codigo del manejador de excepciones 2....
}
```

La estructura es simple. Se coloca dentro de un bloque `try` las instrucciones que podrían lanzar excepciones y las cuales queremos tratar. A continuación se incluyen uno o más bloques `catch` con las excepciones que queremos procesar si producen. Si se produce una excepción que no esté en los catch la excepción no se procesa y el programa termina inmediatamente.

Si se produce una excepción y tenemos un `catch` para ella, se ejecuta el bloque correspondiente al catch. Dentro de este bloque podremos utilizar, si queremos, la información de la excepción que estará almacenada en la variable correspondiente al bloque `catch`.

Faltaría una opción adicional, la opción `finally`, que veremos en futuros bloques ya que está relacionada con la implementación de métodos.

Asimismo, en futuros bloque veremos como crear y lanzar nuestras propias excepciones.

Ejemplo:

Supongamos que tenemos el primer caso de ejemplo visto anteriormente. Un programa que lee dos números y los divide, mostrando por pantalla el resultado. El programa simple sería:

```
Scanner scan = new Scanner(System.in);
System.out.print("Introduzca el dividendo: ");
double dividendo = Double.parseDouble(scan.nextLine());
System.out.print("Introduzca el divisor: ");
double divisor = Double.parseDouble(scan.nextLine());
System.out.println("El resultado de la división es " +
(dividendo / divisor));
```

Este código presenta el problema arriba indicado. En la línea 6 se puede producir un error si se introduce un valor cero. Una nueva versión pero tratando el error de forma elegante podría ser:

```
Scanner scan = new Scanner(System.in);
System.out.print("Introduzca el dividendo: ");
double dividendo = Double.parseDouble(scan.nextLine());
System.out.print("Introduzca el divisor: ");
double divisor = Double.parseDouble(scan.nextLine());
try {
  System.out.println("El resultado de la división es " +
(dividendo / divisor));
} catch (ArithmeticException e) {
  System.out.println("Error. Se ha intentado dividir por cero. El
divisor debe ser distinto de cero. Inténtelo de nuevo más tarde");
```

```
}
```

Esta versión, aunque no realiza la operación, trata el error y muestra un mensaje más amable al usuario. Se deja como ejercicio el hacer el programa de forma que si se intenta dividir por cero se vuelva a solicita o bien el divisor sólo o bien ambos números.

7.- Depuración de programas.

La escritura de un programa es sólo uno de los pasos en la creación del mismo. Tanto o más importante que la codificación es el realizar pruebas a los programas a fin de identificar potenciales problemas y corregirlos antes de entregar nuestro programa al cliente o jefe.

En primer lugar debemos ser cuidadosos durante el desarrollo e intentar preveer los posibles problemas que se puedan presentar. Para hacer esto hay que intentar pensar fuera del molde y no seguir siempre el mismo camino trillado sino intentar hacer cosas inesperadas. Por ejemplo, si un programa solicita un número para un cálculo, ¿Qué ocurre si introduzco texto en su lugar? ¿O un número mal construido (con coma en lugar de punto, por ejemplo)? ¿Cómo se porta el programa? ¿Hace lo que se espera?

De la misma forma hay que intentar actuar como un usuario, a ser posible malicioso, e intentar casos y datos que estén en los límites permitidos, asi como dentro y fuera de ellos. Sólo asi haremos programas robustos o por lo menos sabremos donde están las debilidades de los mismos.

Una vez que comenzamos a realizar las pruebas, detectaremos en la mayoría de los casos comportamientos no deseados. Es en este momento donde el uso de las herramientas de depuración nos facilitará la localización del error como primer paso para corregirlo.

Para ello se emplea un módulo, llamado depurador (debugger) que se asocia a un programa (nuestro programa) en ejecución y permite examinar el funcionamiento real del mismo a fin de detectar el punto en el que el programa deja de hacer lo que nosotros creíamos que iba a hacer.

7.1.- Puntos de ruptura

Un punto de ruptura es un marcador que podemos colocar en una instrucción de nuestro programa y que hace que el mismo se detenga **antes** de ejecutar dicha instrucción. Nos sirve para detener un programa cuando llega a un punto interesante a fin de examinar los datos y ver qué pasos se realizan a continuación.

Una versión más sofisticada permite hacer puntos de ruptura que no detienen siempre la aplicación sino sólo se activan si se cumplen ciertas condiciones, tanto internas como externas a la aplicación.

7.2.- Inspección de datos

Cuando un programa está detenido se nos permite examinar los datos contenidos en las variables y objetos de los que consta nuestro programa a fin de determinar si los valores contenidos se corresponden a los esperados. En caso de discrepancia esto podría indicar un error (o que estabamos equivocados en un principio).

Asimismo se permite la modificación de los datos. Esto puede ser útil en determinadas circunstancias para intentar forzar que nuestro programa haga algo que no iba a hacer con los datos que tenía anteriormente o para forzar un error a fin de ver si se trata adecuadamente.

7.3.- Ejecución paso a paso

Otra característica muy útil de los depuradores es la ejecución paso a paso que nos permite ejecutar las instrucciones una a una, viendo el resultado que producen y cómo van variando los datos

conforme se van ejecutando, asi como para seguir el curso de la ejecución y ver si sigue el camino esperado o se producen cambios de rumbo inesperados. Hay varios tipos de controles de ejecución paso a paso, entre ellos (sin pretender dar una lista exhaustiva):

- Ejecutar la siguiente instrucción por encima. Ejecuta la siguiente instrucción. Si esta es un paso de mensajes la trata como una sola instrucción y pasa a la siguiente cuando termina.
- Ejecutar la siguiente instrucción entrando. Ejecuta la siguiente instrucción. Si ésta es un paso de mensajes continúa la ejecución por el código que trata el mensaje (entramos en el código del mensaje). Útil cuando nosotros hemos hecho también la clase que se usa.
- Ejecutar hasta la salida. Ejecuta el bloque en que estamos hasta que termina.
- Ejecutar hasta aqui. Ejecuta desde el punto donde está actualmente detenido el programa, hasta que llegue a la línea donde está situado el cursor de escritura.

Todos tienen su distinta utilidad según su ocasión y la tarea de depuración que estemos haciendo.

7.4.- ¿Cómo se depura un programa?

Para depurar un programa no hay una receta milagrosa. Aún asi se pueden dar algunas guias generales:

- Hacer un plan de pruebas. Determinar qué casos se van a probar incluyendo datos de entrada y datos esperados de salida.
- Detectar la aparición de un error. Ejecutar el plan de pruebas comparando las salidas esperadas con las obtenidas. Cuando haya alguna discrepancia estamos ante un error.
- Examinar el programa de forma ocular para intentar determinar la línea sospechosa (o el bloque) de ser la causa del error.
- Poner un punto de ruptura en la línea o mejor en una o dos antes para examinar los datos **antes** de llegar a la línea sospechosa.
- Examinar los datos. Si parecen los esperados proceder a la ejecución.
- Examinar los datos **después** de ejecutar la línea sospechosa ¿Son los esperados? Si no hay que intentar enfrentar lo esperado con lo obtenido y ver qué hay que cambiar para que lo obtenido coincida con lo esperado.
- Volver a probar desde el principio. Es bastante usual que la corrección de un error provoque que algo que antes funcionaba, ahora no lo haga. No basta con probar el mismo error y ver si ahora funciona bien sino que hay que volver a probar lo que antes funcionaba para comprobar que sigue funcionando.
- Si pasa todas las pruebas has terminado.

8.- Documentación de programas

Además de realizar nuestro programa, será nuestra obligación documentarlo, por lo menos en la parte que nos corresponda.

Esto es muy importante porque, como ya se ha comentado más de una vez, un programa se lee mucho más que se escribe ya que gran parte de la vida del mismo es mantenimiento que exige leer el código ya realizado. Es por lo tanto de la máxima importancia que esta labor de lectura se vea facilitada lo máximo posible. Para ello hay que seguir una serie de buenas prácticas:

- Seguir unas reglas de estilo que maximicen la uniformidad en el código y la buena organización visual del mismo, facilitando al máximo la lectura.
- Usar comentarios de forma generosa, especialmente en las partes cuyo propósito no queda muy evidente en el mismo código.
- Generar la documentación adicional que se requiera.

8.1.- Documentación interna: comentarios

La documentación interna de una aplicación consiste principalmente en los comentarios. Java no obliga a usar ningún nivel de comentarios (es posible no usar ninguno) pero como programadores debemos documentar el código introduciendo comentarios donde veamos necesario, especialmente en las partes cuyo propósito no quede claro a simple vista.

Por lo menos se debería documentar cada grupo de instrucciones que realicen una labor o tarea bien definida.

Cuando veamos la construcción de clases describiremos técnicas para documentar el interfaz de las mismas de forma que se genere le documentación automáticamente a partir de los comentarios (Javadoc).

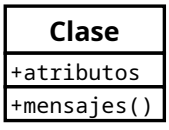
8.2.- Documentación externa: Diagramas de clases UML

Otra forma de documentación que se usa mucho de forma auxiliar o complementaria al código son los diagramas de clases, los cuales indican, de forma gráfica, los objetos o clases que existen y su relación entre ellos.

El más básico es el diagrama de clases que describe, de forma suscinta, lo que describimos a continuación.

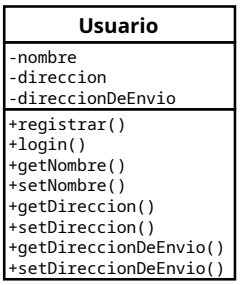
8.2.1.- Clases en UML

Una clase en UML se representa por un rectángulo dividido en tres partes:



En la parte superior se indica el nombre de la clase, en la parte media los atributos y en la parte inferior los mensajes. Un atributo o mensaje se antecede con el símbolo `+` si pertenece al interfaz o con el símbolo `-` si no se muestra en el mismo.

Por ejemplo, una hipotética clase `Usuario` podría ser:



La clase consta de tres atributos (nombre, direccion, direccionDeEnvio). Aqui no hemos puesto los tipos pero se podría añadir. Todos los atributos son privados (están precedidos por -). Después tenemos 8 mensajes, todos públicos. No hemos incluido parámetros aunque una descripción más detallada podría incluirlos, asi como los valores de retorno de los mensajes.

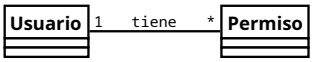
8.2.2.- Relaciones entre clases

Existen diferentes relaciones entre clases. Sin querer entrar en todas las posibilidades, veremos las más comunes.

8.2.2.1.- Asociación

La asociación es una relación entre iguales. Las clases se usan una a otra pero ninguna es más importante ni está subordinada a la otra. Se representa con una línea normal. A veces se emplean multiplicidades en los extremos de la línea para indicar la participación de elementos de un lado con los del otro y se le puede dar un nombre para clarificar su propósito (usualmente es un verbo).

Por ejemplo:

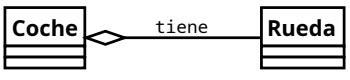


En este caso, instancias de la clase `Usuario` están relacionadas con instancias de la clase `Permiso`. La relación se ha llamado `tiene` y una instancia de la clase `Usuario` puede estar relacionada con cero, una o varias instancias de la clase `Permiso` pero una instancia de la clase `Permiso` sólo puede estar relacionada con una y solo una instancia de la clase `Usuario` (`*` o `n` significan cualquier número, incluido cero).

8.2.2.2.- Agregación

La agregación es una relación asimétrica de la forma todo - parte, esto es un objeto (el todo) está compuesto, en su totalidad o en parte, de otros objetos (las partes). La asimetría viene porque, a veces, las partes no pueden existir si no es conectadas con un todo. La agregación se representa con una línea con un rombo en la parte del todo. El resto es igual a la asociación.

Por ejemplo:



El rombo puede aparecer o no relleno. Se supone que si está relleno la asociación es más fuerte que si no lo está pero no hay un criterio claro de cuando usar uno u otro. Nosotros usaremos siempre el que se ve arriba.

8.2.2.3.- Herencia

Este es un tipo de relación que veremos más adelante pero por ahora baste saber que es una relación de especialización, en la cual una nueva clase extiende (hereda) otra ya existente para añadirle elementos o modificar los ya existentes de forma que la nueva clase reaproveche lo que tiene de común con la antigua, añadiendo o modificando lo que no se corresponda con la realidad de la nueva.

La herencia se representa con una línea con punta de flecha vacía, apuntando en la dirección desde la que hereda hacia la heredada.

Por ejemplo:

En este caso tanto `Perro` como `Gato` heredan de la clase `Animal`.

9.- Resumen

En este bloque hemos aprendido nuevas e importantes técnicas que son básicas para la programación y que se pueden aplicar prácticamente a cualquier tarea de programación y a cualquier lenguaje, aunque nosotros nos hayamos centrado en el lenguaje Java, nuestro lenguaje de referencia.

Asimismo hemos intentado dar respuesta básica a dos tareas que se realizan frecuentemente de forma paralela a la programación: la depuración y la documentación.

10.- Referencias

- Buena página sobre Java en general y sobre las estructuras de control en particular (https://www.ingenieriasystems.com/2015/12/Fundamentos-de-programacion-en-Java-Autor-Jorge-Martinez-Ladron-de-Guevara.html)
- Buena página sobre diagrama de clases UML (https://diagramasuml.com/diagrama-de-clases/)