1. Caso práctico

Unidad Didáctica IX

Caso práctico

Ha llegado el momento de comenzar a utilizar herramientas de programación y ver resultados. ***Víctor*** *lleva mucho tiempo esperando para sentarse al ordenador y probar sus aplicaciones.* ***Carmen*** *le ha dicho que hoy iba a enseñarle el uso del entorno de desarrollo que utilizan habitualmente, y eso le tiene impaciente.*

Víctor *sabe que el entorno de programación es la herramienta más importante para un programador de aplicaciones, ya que como su propio nombre indica, proporciona al programador todo cuanto necesita para desarrollar programas. En este caso ellos utilizan el entorno NetBeans para Java, aunque* ***José*** *le ha comentado que todos los entornos son parecidos y proporcionan herramientas similares, lo que ocurre es que unos hacen las cosas de un modo y otros de forma diferente. Lo que ocurre en estos casos es que trabajar con una aplicación IDE concreta requiere un poco de práctica y cambiar de IDE supone practicar más para familiarizarte con el entorno.*

Unidad Didáctica IX

Introducción.

*Lo que **Carmen** tiene claro, y en eso ha coincidido con **José**, es que lo importante debe ser aprender el lenguaje (básicamente a utilizar las sentencias) porque lo que **SI Andalucía** necesita es un desarrollador, alguien que cree programas ayudándose de la aplicación IDE. Por ello **Carmen** y **José** han decidido planificar la formación de **Víctor** como programador, comenzando con una iniciación a la sintaxis básica de Java y los tipos de sentencias que maneja este lenguaje de programación, todo ello acompañado de una batería de ejercicios imprescindibles para que practique y vaya adquiriendo la soltura necesaria a la hora de codificar programas. Y todo esto será más rápido y mejor si lo hace utilizando NetBeans.*

Tampoco en esta unidad vas a encontrar conceptos totalmente nuevos. Casi todo lo que vamos a ver, una vez más, aparecía en unidades anteriores. Pero también una vez más, trataremos de concretar los aspectos relativos al lenguaje Java.

En unidades anteriores hemos aprendido de forma general los conceptos relativos a algoritmos, datos, tipos, operadores, expresiones, y hemos entrado en los detalles de cómo se plasman esos conceptos en Java.

También hemos visto formas genéricas de representar los algoritmos, las instrucciones y las estructuras básicas de tratamiento, entrando en las consideraciones sobre su uso que resulta necesario tener en cuenta, usemos el lenguaje de programación que usemos. Por eso describíamos los algoritmos de forma genérica usando pseudocódigo o diagramas de flujo, fundamentalmente.

Pero necesitamos ver cómo se plasma todo eso en un lenguaje de programación concreto, de forma que podamos empezar a comprobar que nuestros diseños de algoritmos funcionan correctamente. Y el lenguaje elegido para este módulo profesional ha sido Java. Tenemos que aprender los detalles de Java relativos a las sentencias y al control de la ejecución de las mismas. Ése es el objetivo de esta unidad, **seguir profundizando en la sintaxis de Java, hasta que nos familiaricemos con la sintaxis de las distintas estructuras de control, y sepamos usarlas convenientemente en la resolución de problemas**, al programar los algoritmos.

Al mismo tiempo, al tratarse de una unidad que no introduce demasiados conceptos novedosos, sino detalles de la sintaxis de Java, es el momento adecuado para comenzar a introducir el uso de un Entorno Integrado de Desarrollo o IDE. Hemos elegido **NetBeans**, por varios motivos.

- Usar un entorno de desarrollo facilita y acelera enormemente el proceso de desarrollo de aplicaciones.
- Es un entorno bastante completo, que nos proporciona toda la funcionalidad que necesitaremos en este módulo profesional, incluido el diseño de Interfaces Gráficas de Usuario o GUI.
- Este entorno está programado enteramente en Java, lo que os proporcionará una idea de la potencialidad del lenguaje.
- Está desarrollado por Sun MicroSystem, la empresa creadora de Java, es de código abierto (**open source**) y **se distribuye gratuitamente**, pudiendo descargarse conjuntamente con el JDK de Java, como un único paquete autoinstalable.
- Su facilidad de instalación es otra de sus características importantes.
- Se encuentra disponible para la mayoría de las plataformas y sistemas operativos (Windows, Linux, Solaris y Macintosh).
- Es un entorno con soporte para otros lenguajes de programación, aunque en este módulo profesional no vamos a hacer uso de esa característica.
- Incluye módulos para desarrollo de aplicaciones Java, servicios web y aplicaciones móviles, lo que puede resultar interesante para este módulo profesional, y otros módulos de este ciclo.

Para saber más:
Si deseas conocer más detalles sobre las características del entorno Netbeans, puedes consultar la página siguiente. Eso sí, todo está en inglés, aunque no es necesario tener un alto nivel de idiomas para entender las ideas principales que ahí se exponen.

NetBeans Feature List. **[Versión en Caché]**

(Si tienes problemas para acceder a algún enlace, pulsa en "Versión en Caché" para visualizar una copia de esa página web)
Para descargar el programa Acrobat Reader pulsa aquí.

AUTOEVALUACION

Respecto a las ventajas, a la hora de programar en Java, de usar NetBeans como entorno integrado de desarrollo, podemos indicar:

a) Es un entorno con soporte para otros lenguajes de programación, aunque en este módulo profesional no vamos a hacer uso de esa característica.
b) Usar un entorno de desarrollo facilita y acelera enormemente el proceso de desarrollo de aplicaciones.
c) Es un entorno bastante completo, que nos proporciona toda la funcionalidad que necesitaremos en este módulo profesional, incluido el diseño de Interfaces Gráficas de Usuario o GUI.
d) Todas las anteriores son correctas.

Comprobar

Unidad Didáctica IX

Tipos de sentencias en Java.

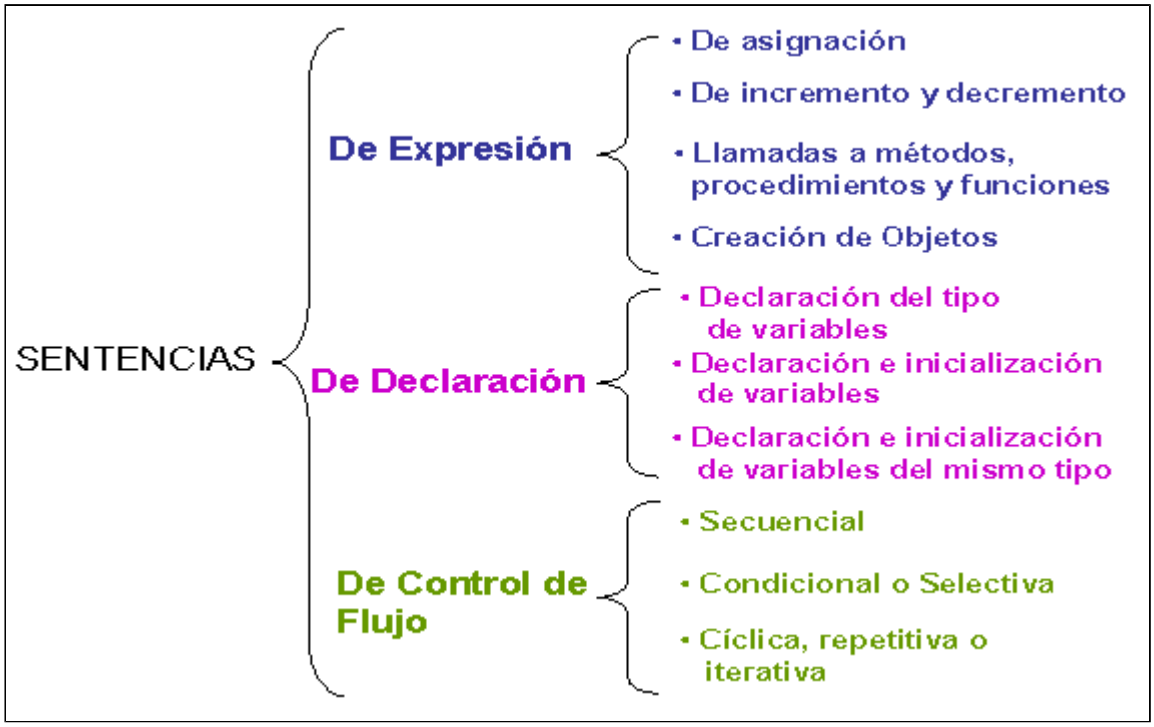
__Víctor__ ya ha visto algunos programas en Java. De hecho los ha compilado en modo texto utilizando "javac", la consola de su equipo, y ejecutándolos (para probarlos) con "java", pero esos programas eran muy simples y lo único que pretendían era que conociera el diálogo entre el usuario y el programa y viera el proceso que se sigue para la compilación de un código fuente en Java. __Carmen__ va a comenzar mostrándole sentencias en Java, su utilidad y cómo incluirlas como código eficiente en un programa.

Comenzamos este apartado con el cuadro resumen que aparecía en la unidad dedicada a operadores, expresiones e instrucciones. Como recordarás, en cualquier lenguaje, los tokens de distintos tipos se unían para formar expresiones, que aunque tienen sentido propio, por sí mismas no indican ninguna acción a realizar por el ordenador. La forma en que esas expresiones pasan a formar sentencias totalmente completas, con sentido pleno y ejecutables por el ordenador, es el tema a tratar en esta unidad.

Es importante recordar que el **carácter de finalización de sentencia en Java es el punto y coma (;)**. Así, todas las sentencias terminarán en punto y coma.

Viendo este esquema recordamos que hay tres tipos básicos de sentencias:

- **Sentencias de expresión**. Están formadas por una expresión seguida del carácter; (punto y coma).
- **Sentencias de declaración**. La declaración de una variable, en la que se le asocia al identificador de la variable un tipo, y opcionalmente un valor, que genera una sentencia, al añadirle el símbolo de terminación.
- **Sentencias de control de flujo**. Se encargan de "contener" a las otras sentencias del programa, de forma que se indique el orden en que se van a ejecutar esas sentencias, y bajo qué condiciones. Pueden considerarse como sentencias estructurales dentro del programa.



AUTOEVALUACION

Señala la afirmación correcta. Respecto a las sentencias de Java, podemos decir que:

a) Las sentencias de expresión son: de asignación, de creación de objetos, de incremento y decremento y llamadas a métodos, procedimientos y funciones.
b) Las sentencias de expresión son: secuencial, cíclica, repetitiva y condicional.
c) Todas las expresiones dan lugar a sentencias válidas sin más que añadirles un punto y coma.
d) Todas las anteriores son correctas.

Comprobar

Señala la afirmación correcta. Respecto a las sentencias de Java podemos afirmar que:

a) Las sentencias de control de flujo son: de incremento y decremento y llamadas a métodos, procedimientos y funciones.
b) Las sentencias de control de flujo son: secuencial, cíclica, repetitiva y condicional.
c) El principal grupo de expresiones de asignación se obtienen añadiendo punto y coma a sentencias de expresión.
d) Todas las anteriores son correctas.

Comprobar

Unidad Didáctica IX

Sentencias de expresión.

No existe un programa que no tenga una sentencia de este tipo. Probablemente se trate de las sentencias que menos cambien de un lenguaje a otro, ya que básicamente permiten el intercambio de datos entre variables. ***Víctor*** *no conoce muchos lenguajes de programación, así que no le da mucha importancia, pero* ***Carmen*** *le explica lo importante que es poder escribir una sentencia que equivale a varios lenguajes (podría ser interesante unificar la sintaxis de varios lenguajes de programación). De todos modos actualmente para aprender a utilizar un nuevo lenguaje, en la mayoría de los casos basta con buscar en un manual las sentencias de cada tipo, si ya sabes programar, claro.*

La base de cualquier aplicación son las variables de datos y los objetos sobre los que se van a aplicar los métodos que sintetizan una determinada tarea. ***Carmen*** *insiste a* ***Víctor*** *que este tipo de sentencias son el eje de todo programa y tiene que conocerlas bien para utilizarlas correctamente. Para ello le identifica este tipo de sentencias en algunos de los programas que ya han trabajado, pero le aclara que no es lo mismo ver una aplicación terminada y funcionando, que intentar crearla de la nada, para esto último es imprescindible conocer bien el lenguaje, las posibilidades que nos proporciona y mucha imaginación.*

No todas las expresiones dan lugar a sentencias válidas sin más que añadirles un punto y coma. Por ejemplo a >b; no tiene ningún sentido por sí misma, no indica ninguna tarea a realizar. Por tanto no basta añadir punto y coma a una expresión lógica para formar una sentencia o instrucción válida.

Si miras el esquema anterior puedes ver que el principal grupo de sentencias de expresión se obtienen añadiendo punto y coma a expresiones de asignación.

Entre las expresiones de asignación incluimos a las formadas por el operador de asignación básico, (**=**) o por cualquiera de los operadores de asignación combinados (**+=, -=, *=, /=, %=**, principalmente). También a las formadas por los operadores de incremento o decremento (**++**, **--**), que llevan una asignación implícita.

Todas esas expresiones son sentencias al añadirles el punto y coma, ya que indican una acción a realizar, que consiste en modificar el valor almacenado en una variable.

En el ejemplo de Gestión de Depósitos de la unidad 7 podemos encontrar ejemplos abundantes de sentencias de expresión de todos los tipos que aparecen en el Esquema anterior.

Unidad Didáctica IX

Ejemplos de sentencias de asignación.

Son numerosas las situaciones en las que necesitamos asignarle valor a una variable.

En la unidad 7 habíamos visto un ejemplo de un pequeño programa para gestionar un par de depósitos. En la clase **Deposito** de ese ejemplo, había un método especial que se llamaba también **`Deposito( )`**, igual que la clase. Los métodos que se llaman igual que la clase son los constructores, y son los encargados de crear los objetos de la clase en cuestión. Así, el método **`Deposito( )`** de la clase **Deposito** sirve para crear objetos concretos o instancias de la clase (depósitos concretos, con una capacidad, un nombre, para almacenar una sustancia determinada, etc.)

Concretamente en este caso, se le pasan como parámetros los valores iniciales con los que se va a crear ese objeto, y posteriormente, se le asignan esos valores a sus variables. Todas las sentencias que contiene el método son sentencias de asignación que usan el operador de asignación básico **`=`** , salvo la última, que usa el operador de incremento **`++`**. En todos los casos, se trata de asignar a la variable correspondiente del objeto depósito para el que se ejecute el método, el valor correspondiente de entre los que se le han pasado como parámetros.

Para el caso de la sentencia de incremento, se trata de sumarle uno al número de depósitos creados, ya que **`totalDepósitosCreados`** es una variable estática, es decir, que existe una única copia para toda la clase. No es una variable que forme parte de cada objeto, sino que es un valor que se usa para la clase, justamente para contar los objetos de esa clase que se van creando.

Destacamos en negrita y con un tamaño un poco mayor las sentencias de asignación.

```
public Deposito(String nombre, String sustancia, int capacidad){
        nombreDeposito=nombre;
       sustanciaQueContiene=sustancia;
        cantidadQueContiene=0;
        capacidadDeposito=capacidad;
        totalDepositosCreados++;
 }
```

Otro ejemplo en el que se usa un operador de asignación combinado es el método **`sacarDeDeposito()`**, también en la clase Deposito. En él destacamos la sentencia de asignación, que indica que se le reste la cantidad pasada como parámetro al valor de la variable **`cantidadQueContiene`** del depósito para el que se ejecute el método.

Destacamos en negrita y con un tamaño un poco mayor la sentencia de asignación que usa el operador combinado **`-=`**.

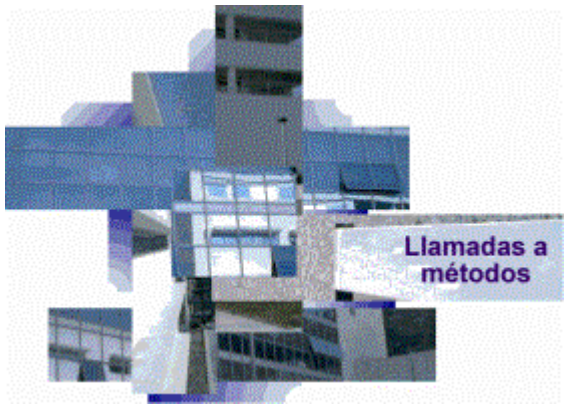
```
public boolean sacarDeDeposito(int cantidad){
   boolean correcto=false;
   if(cantidad >=0){
   if(cantidadQueContiene>=cantidad){
     correcto=true;
             cantidadQueContiene-=cantidad;
   }else{
      System.out.println("No se pueden sacar "+ cantidad +" litros de "
                         +sustanciaQueContiene+  ".
                         El deposito solo contiene "+cantidadQueContiene+" litros.");
   }
   }else{
     System.out.println("No se admiten cantidades negativas");
   }
   return correcto;
```

```
    }
```

Unidad Didáctica IX

Ejemplos de sentencias de llamadas a métodos.

Seguimos con el esquema de los tipos de sentencias. ¿Qué otro tipo de expresiones pueden formar por sí mismas sentencias?Aún quedan dos, las llamadas a métodos y la creación de objetos. En este apartado nos ocuparemos de las llamadas a métodos.



Cualquier llamada a un **método** (también llamados mensajes, en la jerga de programación orientada a objetos, o funciones y procedimientos, en general.) tiene por sí misma un sentido pleno. Le indica al ordenador que sustituya esa llamada por todas las instrucciones que contiene ese método en su definición, y que las ejecute. Por tanto, **añadir punto y coma a una llamada a un método es una sentencia en Java, e indica claramente una tarea a realizar.**

Continuando con el ejemplo de Gestión de Depósitos de la unidad 7, tenemos muchos ejemplos de llamadas a métodos. Nos vamos a fijar en la clase GestionDepositos, que nos muestra un menú, y nos da a elegir una opción. Dependiendo de la opción elegida, se llama a un método o a otro, que es el que se encarga de realizar la tarea adecuada.

Fíjate en que el uso de métodos que son invocados es a la vez una forma (aunque no la única) de hacer programación modular, y de aplicar el diseño descendente para solucionar nuestro problema.

En concreto, nos fijamos en la sentencia switch que permite seleccionar el método a ejecutar una vez que hemos elegido la opción que deseamos usar. Destacamos en negrita y en un tamaño un poco mayor las llamadas a métodos que aparecen.

```
...
operacion = ES.leeN°("Elige una opcion del menu: ");
...
/* Una vez elejida la operación, ejecutamos el método adecuado para realizarla */
      switch (operacion) {
        case 0:
          System.out.println("EL PROGRAMA HA FINALIZADO. ADIÓS. ");
        case 1:
          listadoDepositos();
          break;
        case 2:
          llenarElDeposito();
          break;
        case 3:
          meterAlgoEnDeposito();
          break;
        case 4:
          vaciarElDeposito();
          break;
        case 5:
          sacarAlgoDeDeposito();
          break;
      }
```

Algunos de esos métodos los hemos definido nosotros mismos, y otros nos los proporciona el propio lenguaje. Éste es el caso del método **`println()`**. Este método nos lo proporciona Java, pero para llamarlo, tenemos que hacerlo a través del objeto **`out`** de la clase **`System`**, que representa el dispositivo de salida estándar, y que suele ser la pantalla.

Por tanto, toda la línea es una única llamada al método **`println()`** indicando también al compilador dónde

puede encontrar la definición de ese método (las sentencias que habrá que ejecutar en lugar de la llamada), ya que no aparece en la propia clase **GestionDepositos.**

Algo parecido ocurre con la llamada al método **leeNº().** Como no está definido en la clase GestionDepositos, debemos indicar en la llamada dónde debe buscar el compilador la definición del mismo para sustituir la llamada por el conjunto de sentencias de la definición y ejecutarlas. Por eso tenemos que escribir la llamada como **ES.leeNº()**, ya que el método está definido en la clase ES de nuestro ejemplo.

Las demás llamadas corresponden a métodos que sí están definidos en la clase desde la que se llaman, **GestionDepositos**, por lo que basta con escribir directamente su nombre.

Por otro lado, la llamada a un método puede incluir a su vez llamadas a otros métodos. Es el caso de **sacarAlgoDeDepostio()**, que llama a su vez a los métodos **seleccionarDeposito()** y **sacarDeDeposito()** definidos en la propia clase, y a los métodos **println()** y **leeNº()** definidos en otras clases.

En negrita y con un tamaño de letra un poco mayor destacamos las llamadas a otros métodos que se hacen desde el método **sacarAlgoDeDeposito()**

```
public static void sacarAlgoDeDeposito(){
   boolean operacionFinalizadaCorrectamente=false;
   int cantidad=0;
   int dep = seleccionarDeposito();
       if (dep == 0) {
         System.out.println("No es posible sacar nada en un deposito si no se selecciona
       }
       else {
         cantidad = ES.leeNº("¿Que cantidad desea sacar?");
         if (dep == 1) {
           d1.sacarDeDeposito(cantidad);
         }
         else {
           d2.sacarDeDeposito(cantidad);
         }
    }
    if (operacionFinalizadaCorrectamente){
      System.out.println(" Se han sacado "+cantidad+" litros del deposito" );
    }else{
      System.out.println(" NO FUE POSIBLE SACAR "+cantidad+" litros del deposito" );
    }
 }
```

Unidad Didáctica IX

Ejemplos de sentencias de creación de objetos.

¿Recuerdas de qué tipo de sentencias se trata? Si miras el esquema del apartado 2 verás que es el último tipo de sentencias de expresión que nos queda. ¿Y a partir de qué expresiones se formarán? Evidentemente a partir de expresiones de creación de objetos, que son las que involucran al operador **new()**

El operador **new()** tiene por su sintaxis más el aspecto de un método que de un operador, pero es un operador.

En la unidad anterior ya vimos con detalle la forma habitual de este operador:

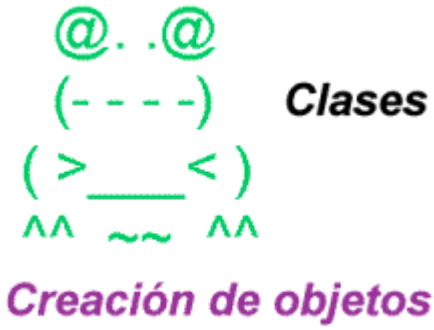
```
<Nombre de la clase>  <nombre de la variable objeto> = new <llamada al constructor de la
        misma clase>
```

Y en el ejemplo de Gestión de Depósitos, la clase **GestionDepositos** era la encargada de crear un par de depósitos y de operar con ellos. Las sentencias de creación de esos depósitos eran como las que siguen:

```
Deposito d1 = new Deposito("DEPOSITO1", "AGUA", 200);
Deposito d2 = new Deposito("DEPOSITO2", "ACEITE", 500,250);
```

En la unidad anterior ya explicábamos las acciones que implicaban esas dos sentencias de creación de objetos:

- Se busca espacio libre donde alojar el objeto de la clase que se le indique al operador **new()**.
- Se crea y se inicializa el objeto.
- Se guarda su dirección de memoria en una variable del tipo del objeto (realmente una referencia al objeto recién creado).



Realmente, aunque ésa es la forma más normal de usar dentro de una sentencia el operador **new()**, **ninguna de las dos sentencias anteriores es una sentencia de creación de objetos. Al menos no son sólo eso**.

Si nos fijamos, realmente son sentencias de asignación, ya que asignan la dirección de memoria donde se ha creado y alojado el objeto a una variable (una referencia)

Con esto, lo que queremos aclarar es que resultaría perfectamente posible hacer la sentencia de creación sin hacer una asignación:

```
new Deposito("DEPOSITO1", "AGUA", 200);
```

Es una sentencia de creación de objetos pura. Realmente el operador new crearía un objeto depósito llamado DEPOSITO1, para contener AGUA y con una capacidad de 200 litros.

El problema es que si no asignamos la dirección de memoria en la que lo ha alojado a alguna variable (referencia) no tendremos forma de acceder a él y no podremos utilizarlo en el programa.
Por eso rara vez ejecutaremos una "sentencia de creación de objetos pura". Lo normal es que aparezca integrada en una sentencia de asignación.

Java dispone de un proceso que se ejecuta en segundo plano y que se encarga de comprobar si los objetos que hay en memoria están referenciados o no. Si ninguna referencia apunta a un objeto, y dado que en ese

caso no podremos llegar a él para usarlo, lo considera basura, y lo elimina automáticamente, liberando la memoria que ocupaba, sin que el programador tenga que preocuparse de ello. Ese proceso se llama "recolector de basura" o "garbage collector". Así, para eliminar de la memoria un objeto basta con hacer que todas sus referencias dejen de "apuntarlo" o referenciarlo, y tarde o temprano el recolector de basura se activará y lo eliminará definitivamente.

PARA SABER MÁS:
En este enlace encontrarás información sobre qué es el recolector de basura en un entorno de desarrollo en Java, su funcionamiento y las ventajas de su uso.
***TUTOR DE JAVA*. [Versión en Caché]**

(Si tienes problemas para acceder a algún enlace, pulsa en "Versión en Caché" para visualizar una copia de esa página web)
Para descargar el programa Acrobat Reader pulsa aquí.

AUTOEVALUACION

Respecto al recolector de basura de Java, señala la afirmación correcta.

- a) Para eliminar de la memoria un objeto basta con hacer que todas sus referencias dejen de "apuntarlo" o reverenciarlo.
- b) Para eliminar de la memoria un objeto basta con hacer que todas sus referencias se actualicen y activar el recolector basura.
- c) El recolector de basura borra todas las variables y objetos que no han sido usados recientemente liberando así memoria.
- d) Todas las respuestas anteriores son ciertas.

Comprobar

Unidad Didáctica IX

Sentencias de declaración.

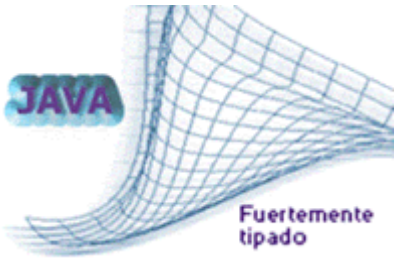
*Parece que **Víctor** ha asimilado bastante bien las sentencias de expresión. Se le nota que tiene ganas de aprender, presta mucha atención y permanece concentrado ante las explicaciones de su compañera.*

***Carmen** cree que ha llegado el momento de que empiece a definir las variables y objetos que va a utilizar en un programa, para después declararlos y utilizarlos con garantías. Hasta el momento ha sido ella quien le ha dicho qué variables debe utilizar y le ha dado el código para declararlas en el programa, pero un programador no puede depender de otro que le proporcione las variables, así que debe tener absoluta autonomía ante cualquier problema. Esto a **Víctor** le estimula y le gusta la idea de manejarse por sí solo ante cualquier programa.*

Pero en un programa haremos más cosas, no todo va a ser asignar valores a variables. Y además, también deberemos declarar esas variables ¿Cómo se forman las sentencias de este otro tipo?

Realmente basta con indicar el tipo de una variable seguido de su identificador y terminar con un punto y coma, como cualquier otra sentencia.

El compilador de Java siempre que encuentra un tipo seguido de un identificador, interpreta que se trata de una declaración.



Las declaraciones consisten básicamente en indicar, la primera vez que aparece en el programa una variable, cuál va a ser su tipo. Esto ayuda al compilador a saber cuanto espacio debe reservar en memoria para esa variable, y permite que se pueda simplificar las comprobaciones que debe hacer el compilador, ya que toda variable que usemos debe ser previamente declarada.

Java es, según lo dicho, un lenguaje fuertemente tipado. Esa cualidad, lejos de ser un inconveniente como puede parecer a primera vista, es una garantía de seguridad, y de que los valores que nos vamos a encontrar en una variable son los esperados, y nunca cualquier otra cosa.

Podemos construir sentencias de declaración de tres tipos:

- **Declaración simple de una variable `int a;`**
- **Declaración e inicialización de una variable `int a = 0;`**
- **Declaración e inicialización de varias variables del mismo tipo simultáneamente. `int a = 0, b, c = 5;`**

AUTOEVALUACION

Señala la afirmación correcta. Respecto a las sentencias de declaración en Java, podemos afirmar que:

a) Las declaraciones consisten básicamente en indicar, la primera vez que aparece en el programa una variable, cual va a ser su tipo.

b) La sentencia int a = 0; es de tipo declaración simple de una variable.

c) El que Java sea un lenguaje fuertemente tipado es un inconveniente a la hora de declarar una variable.

d) Todas las anteriores son falsas.

Comprobar

Sentencias y control de ejecución en java.

Unidad Didáctica IX

Ejemplos de sentencias de declaración.

Al igual que la asignación, te resultará más que difícil hacer cualquier programa que haga algo más que escribir un texto por pantalla sin usar ninguna sentencia de declaración. ¿Por qué? Sencillamente porque como hemos dicho en el apartado anterior, cualquier variable que usemos en Java debe estar declarada previamente. Luego no podemos hacer nada con la variable si no la hemos declarado.

En nuestro ejemplo de Gestión de Depósitos, hay numerosos ejemplos de sentencias de declaración. En ellos, además de la simple declaración de tipo y nombre de la variable, aparecen algunos modificadores, que alteran con matices importantes el significado de esa declaración, pero que no alteran el hecho fundamental de que la sentencia está indicando que se va a usar una nueva variable a la que se le da nombre y se le asigna un tipo. Veamos algunos de esos ejemplos en los apartados siguientes.

Unidad Didáctica IX

Declaración simple de una variable.

En la clase Deposito, cuando tenemos que definir qué estructura tiene un Deposito, declaramos varias variables miembro de objeto, que son las variables que forman parte de cada objeto, y que tomarán valores distintos para objetos distintos. En el ejemplo, cada depósito tendrá un nombre distinto, por ejemplo.

```
final String nombreDeposito;
final String sustanciaQueContiene;
private int cantidadQueContiene=0;
final int capacidadDeposito;
```

En este caso, además del tipo y del nombre de la variable, aparecen delante otras palabras, que actúan como modificadores de la declaración. Concretamente final indica que se trata de una constante, es decir, que una vez que le asignemos un valor ya no podremos modificarlo. En este ejemplo, al crear un depósito

- Le asignamos un nombre.
- Indicamos la sustancia que vamos a almacenar.
- Le asociamos una capacidad. .

Ninguno de esos valores podrá ser modificado después de haber creado el depósito.

Parece sensato que sea así en el caso de la capacidad. Un depósito tendrá la capacidad que se le ha dado al construirlo, y no podemos agrandarlo o reducirlo a conveniencia.

No es tan evidente en el caso del nombre. ¿Por qué no vamos a poder cambiarle el nombre a un depósito?

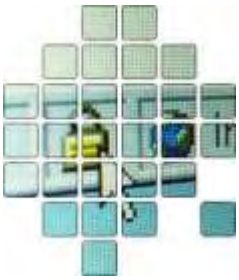
Es una decisión del programador que puede ser fruto del análisis del problema. Imagina que en los depósitos se guardan sustancias peligrosas, como explosivos o uranio enriquecido, de las que la empresa necesita tener un control estricto. Por ejemplo, necesita mantener un registro o historial de qué operaciones se han realizado con un depósito concreto, y cambiarle el nombre al depósito puede hacer inútil la información guardada en ese histórico. No tendríamos certeza de a qué depósito nos referimos. Algo parecido sucede con la sustancia que contiene. No parece apropiado guardar aceite de oliva virgen extra en un depósito que anteriormente contenía cianuro pótásico o veneno matarratas.

En cualquier caso, la decisión de que sean constantes o variables, vendrá justificada por el análisis del problema.

En el caso de **`cantidadQueContiene`**, usamos el modificador **`private`**. Más adelante veremos con detalle estos modificadores, pero adelantamos aquí lo que significa:

- Indica que a esta variable sólo se le va a poder modificar el valor desde la propia clase en la que se ha definido o a través del uso de métodos definidos en la propia clase.
- Sólo podremos modificar su valor desde otras clases a través de métodos públicos de la clase **`Deposito`**.
- De esta manera, cualquier comprobación de seguridad que necesitemos realizar, podemos incluirla en esos métodos públicos.
- Ninguna aplicación que use la clase **`Deposito`** podrá modificar el contenido de un depósito concreto sin respetar escrupulosamente las comprobaciones de seguridad.

En nuestro ejemplo, el método **`sacarDeDeposito()`** es público (**`public`**), y se encarga de **impedir que podamos sacar de un depósito más cantidad de la que contiene**.



Pero también podríamos haber comprobado que quien da la orden es un usuario identificado con una contraseña y autorizado a hacerlo. Algo así habríamos tenido que hacer, por ejemplo, si nuestro depósito contuviera uranio enriquecido, ¿no?

Por lo demás, esos modificadores no hacen más que aportar matices, aunque importantes, a la declaración de la variable. Lo básico de las dos primeras sentencias es que indican que se trata de constantes de tipo String (cadena de caracteres) y las dos últimas que se trata de una variable y una constante de tipo int (entero)

Unidad Didáctica IX

Declaración e inicialización de una variable.

En el ejemplo de la Gestión de Depósitos, concretamente en el método **`sacarAlgoDeDeposito()`** de la clase **`GestionDepositos`**, aparecen las siguientes tres sentencias de declaración:

```
boolean operacionFinalizadaCorrectamente=false;
int cantidad=0;
int dep = seleccionarDeposito();
```

En todas ellas, además de declarar la variable como perteneciente a un tipo, se le asigna un valor inicial, se inicializa la variable.

En la **primera** sentencia, se declara la variable de nombre **`operacionFinalizadaCorrectamente`** como de tipo **`boolean`**, y se le asigna el valor inicial de **`false`**.

En la **segunda** sentencia se declara la variable de nombre **`cantidad`** como de tipo **`int`**, y se le asigna el valor inicial de **`0`**.

En la **tercera**, se define la variable de nombre dep como de tipo **`int`** y se le asigna como valor inicial el devuelto por el método **`seleccionarDeposito()`** después de ejecutarse. Es decir, se ejecuta la llamada al método **`seleccionarDeposito()`**, que devolverá un número entero correspondiente al deposito que se ha seleccionado para trabajar, y ese valor entero se asigna a la variable **`dep`** que se está declarando como de tipo **`int`**.

Unidad Didáctica IX

Declaración e inicialización de varias variables del mismo tipo simultáneamente.

En el ejemplo de Gestión de Depósitos no tenemos más que una sentencia de este tipo, aunque sólo declara dos variables del mismo tipo simultáneamente. No se inicializan en este caso, pero también podrían ser inicializadas, una o ambas. La sentencia en cuestión aparece al comienzo de la clase GestionDepositos.

```
static Deposito d1, d2;
```

Se definen dos variables de nombre **d1** y **d2** y se indica que ambas van a ser de tipo **Deposito**. Eso se consigue indicando los nombres de todas las variables que van a ser de ese tipo separados por comas, formando una lista. Aparece también un nuevo modificador, que es la palabra **static**. Se usa para indicar que son variables miembro de clase., no de objeto.

En la práctica, queremos decir que d1 y d2 no son campos o valores que contengan los objetos de la clase **GestionDepositos**. ¡Ni siquiera tenemos intención de crear objetos de esa clase, que sirve sólo para gestionar los depósitos!

Lo que indicamos es que para la clase se definen esas dos variables, que van a ser globales a todos los métodos de la clase, es decir que se definen para poder ser usadas por toda la clase y desde dentro de cualquier método de la clase.

Y aunque no aparece ninguna sentencia en la que además de declarar varias variables del mismo tipo se les asigne valor, perfectamente podríamos haber reescrito el siguiente código:

```
int cantidad = 0;
int dep = seleccionarDeposito();
```

en una sola sentencia como:

```
int cantidad = 0 , dep = seleccionarDeposito();
```

Tú mismo puedes hacer esa modificación en el código del ejemplo y comprobar que todo sigue funcionando exactamente igual.

Unidad Didáctica IX

Sentencias de control de flujo.

__Carmen__ está convencida que __Víctor__ será un buen programador por el interés que muestra en aprender y la ilusión que tiene por conocer los detalles de la programación de ordenadores. Recuerda que uno de sus profesores le decía que la cualidad más valiosa para un estudiante es el interés por lo que estudia, de hecho uno de sus compañeros tenía en casa lo último en ordenadores, con todos los programas y todo cuanto podía ser útil para estudiar el módulo profesional, sin embargo no consiguió el título porque no estaba motivado.

En cambio ella tuvo que esperar hasta el segundo trimestre del primer curso para poder comprarse el ordenador con lo que había conseguido trabajando en unos grandes almacenes de su localidad durante las vacaciones de Navidad. De este modo, trabajando, consiguió las mejores notas de su promoción y ahora se dedica a algo que le apasiona.
Ahora tiene que enseñar a __Víctor__ cómo controlar la ejecución de las diferentes sentencias del programa, de modo que realice determinadas acciones bajo condiciones muy variadas, respondiendo así a diferentes estados o situaciones de la vida real. Esto lo realizan las sentencias condicionales y bucles, que en Java se utilizan de forma muy sencilla.

Básicamente hasta ahora hemos visto cómo expresar mediante sentencias la manera de indicarle al ordenador que debe definir y crear variables de un determinado tipo, y hemos visto cómo poder asignarle valores a esas variables.

Esos valores también hemos visto que se pueden obtener de literales, de expresiones del mismo tipo que la variable, o de llamadas a métodos, que actúan como "funciones" generando como resultado de su ejecución un valor devuelto del tipo de la variable. La llamada al método se sustituye por el valor que devuelve ese método.

Pero

- ¿Nos basta con eso?
- ¿Podemos representar así toda la variedad de situaciones que nuestros programas deben afrontar?
- ¿Sabemos ya expresar en Java las sentencias que nos permitirán establecer el orden en que se van a ejecutar las variables?
- ¿Disponemos ya de una sentencia que nos permita decidir entre varias sentencias alternativas cuál es la que debemos ejecutar?

Precisamente eso es lo que vamos a introducir en este apartado. Vamos a ver la **sintaxis en Java de las sentencias de control del flujo de ejecución de los programas.**

Como recordarás de la unidad 5, dedicada a la programación estructurada, esas sentencias son básicamente las que nos van a permitir representar la secuenciación de sentencias, la ejecución condicional de sentencias, y la ejecución cíclica de sentencias: Las tres estructuras básicas de la programación estructurada.

Naturalmente, también mencionaremos la existencia de sentencias de salto incondicional, pero intentaremos acostumbrarnos a hacer nuestros programas sin hacer uso de ellas. Recuerda que podían ser útiles, pero que son desaconsejables en condiciones normales. Entre los objetivos de este módulo profesional no figura enseñaros a hacer programas extraños, así que las condiciones normales serán lo habitual, y el uso de sentencias de salto incondicional, prácticamente nulo.

Unidad Didáctica IX

Ejecución secuencial.

¿Cómo indicarle al ordenador, en cualquier lenguaje, el orden en que debe ejecutar varias sentencias que deben ir unas detrás de otras? Tal y como se lo indicarías a una persona, parece lo más sensato: "Escribiendo esas sentencias una detrás de otra exactamente en el orden en que deben ejecutarse."

¿Y debo tener algo más en cuenta en el caso de Java?

Sí. En Java pueden escribirse varias sentencias, incluso todo el programa, en una sola línea, aunque lo recomendable es escribir cada una en una línea y usar indentación. También es posible, y ocurre frecuentemente, que una sola sentencia ocupe varias líneas del programa. Pero al permitir esas posibilidades nos obliga a delimitar donde termina y donde empieza cada sentencia ¿Cómo?

- **En Java todas las sentencias se terminan con un carácter de punto y coma (;)**
- **Existe la sentencia nula, que es el carácter punto y coma.** En cualquier sitio donde la sintaxis de Java admita una sentencia, también podremos poner la sentencia nula, es decir, simplemente un punto y coma, para indicarle que no haga nada. No obstante, cualquier programa puede hacerse sin necesidad de usar la sentencia nula.
- **Se pueden definir bloques de sentencias.** En cualquier sitio donde la sintaxis de Java admita una sentencia, se puede colocar todo un bloque de sentencias, que se ejecutarían como una sola en el orden que se indique, sin más que encerrarlas entre llaves

```
{sentencia1; sentencia2;...; sentenciaN;}
```

O bien

```
{
sentencia1;
sentencia2;
...;
sentenciaN;
}
```

El ejemplo siguiente muestra un programa en el que se ejecutan secuencialmente una serie de sentencias que realizan operaciones matemáticas.

☐ **Descarga el archivo zip con OperacionesMatematicas.java**

Evidentemente, se puede conseguir el mismo resultado sin que sea totalmente imprescindible escribir las instrucciones en el orden exacto. No todas las sentencias tienen un efecto visible, y por tanto podemos realizarlas en distinto órden, siempre y cuando el resultado final sea el mismo.

En el siguiente ejemplo, presentamos la versión 2 del programa anterior, pero declarando al principio todas las variables que se van a usar. Algunos lenguajes exigen que se haga así, pero en Java basta con que cualquier variable se haya declarado antes de ser usada por primera vez, da prácticamente igual donde se haga esa declaración.

☐ **Descarga el archivo zip con OperacionesMatematicas_V2.java**

Incluso es posible hacer otra tercera versión en la que vamos a agrupar las sentencias en varios bloques:

- Declaraciones
- Entrada
- Proceso
- Salida

☐ **Descarga el archivo zip con OperacionesMatematicas_V3.java**

Como puedes comprobar, las únicas diferencias están en la forma de escribir las operaciones, pero el resultado es siempre el mismo. Lo que sí tenemos que tener presente es que en los tres casos:

- Las declaraciones deben ir siempre antes de usar una variable.
- La primera vez que se usa una variable debe ser para asignarle un valor (inicializarla)
- No podremos hacer nada con una variable (por ejemplo, escribir su valor en pantalla) hasta que haya sido inicializada.

AUTOEVALUACION

Respecto a las sentencias de control de flujo en Java, señala la afirmación correcta:

- ○ a) En Java todas las sentencias se terminan con un carácter de punto y coma (;)
- ○ b) En cualquier sitio donde la sintaxis de Java admita una sentencia, también podremos poner la sentencia nula, es decir, simplemente un punto y coma, para indicarle que no haga nada. No obstante, cualquier programa puede hacerse sin necesidad de usar la sentencia nula.
- ○ c) En cualquier sitio donde la sintaxis de Java admita una sentencia, se puede colocar todo un bloque de sentencias, que se ejecutarían como una sola en el orden que se indique, sin más que encerrarlas entre llaves.
- ○ d) Todas las anteriores son correctas.

Comprobar

Unidad Didáctica IX

Ejecución condicional.

¿Recuerdas los usos de la estructura condicional? En los programas, como en la vida, hay que tomar decisiones, y elegir una entre dos o más alternativas, según las circunstancias.

En programación, las "circunstancias" que nos permiten tomar un camino u otro, ejecutar unas instrucciones u otras en definitiva, son expresiones condicionales, que tomarán el valor verdadero (true) o falso (false).

A Carmen le han encargado un programa, que entre otras cosas debe calcular el sueldo diario de un empleado. Para ello parece adecuado dividir el sueldo mensual entre el número de días trabajados por el empleado en el mes.

Suponemos que el sueldo mensual es conocido, y que para la categoría profesional de los empleados a los que se les va a aplicar el cálculo es para todos ellos igual al salario mínimo incrementado en un 15%.

Aunque nosotros vamos a hacer el código introduciendo manualmente los datos de cada trabajador (concretamente el número de días trabajados en el mes). En la realidad, Carmen debería hacer el programa para que procesara un fichero de trabajadores del que obtendría la información de todos los empleados, y haría el cálculo indicado para cada uno de ellos. ¿Qué ocurriría si un empleado no trabajó ningún día? Al intentar dividir el sueldo entre 0, daría un error, y el programa abortaría si la división es entera, o daría Infinity (constante Double que representa el infinito matemático) como resultado, si la división es real.

La división por cero no está definida, y en principio debemos evitar que el programa permita hacer este tipo de operaciones.

En el ejemplo que sigue, Carmen ha resuelto el problema sin comprobar si el número de días trabajados es igual a cero, y en este caso el programa aborta. Compruébalo tú mismo ejecutando el código del ejemplo.

☐ **Descarga el archivo zip con SueldoMedioDiario.java**

Si has introducido 0 como días trabajados, el resultado es un mensaje como el siguiente, y que el programa aborta.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Que viene a decir

"Excepción en el proceso main. Excepción Aritmética: División por cero".

Sin embargo, si las variables se definen de tipo double, para que la división sea real, el programa funciona, pero produce una salida rara, al indicar que el sueldo medio diario es Infinity.

Ejecuta tú mismo el código modificado, y compruébalo.

☐ **Descarga el archivo zip con SueldoMedioDiarioReal.java**

Para evitarlo, Carmen necesita comprobar el número de días trabajados.

- Si es positivo, hará la división.
- Si es cero, mostrará un mensaje diciendo que el empleado no trabajó ese mes
- Si es negativo, mostrará un mensaje de error.

¿Cómo le indica Carmen al programa que compruebe esas condiciones y que realice la tarea adecuada en cada caso?

Haciendo uso de las **estructuras de control condicionales**. Empezaremos viendo la sintaxis de esas estructuras en Java, y posteriormente las usaremos para resolver el problema de Carmen.

Sentencias y control de ejecución en java.

Unidad Didáctica IX

Condicional simple. Sentencia if.

Imagina una situación en la que existen dos tareas distintas a realizar, y se realizará una u otra dependiendo de que se cumpla o no una condición. Es justamente la situación en la que aplicaremos la sentencia if.

Su sintaxis, junto a la explicación de su funcionamiento, es alguna de las siguientes:

Forma completa de la sentencia if con dos alternativas

```
if (expresión-lógica)
  sentencia1;
else
  sentencia2;
```

Forma simple de la sentencia if con una única alternativa.

```
if (expresión-lógica)
    sentencia1;
```

- Si al evaluar la expresión-lógica resulta verdadera, se ejecutará la sentencia1, y se continuará con la sentencia que haya después del if.
- Si por el contrario expresión-lógica resulta falsa, se saltará sin ejecutarse la sentencia1 y se ejecutará la sentencia2, continuándose igualmente con la siguiente
- La cláusula else de la sentencia if no es obligatoria. Sólo se incluye si también hay que hacer algo en caso de que la expresión lógica sea falsa. Si la sentencia if no tiene parte else, y la expresión resulta falsa, sencillamente se continúa con la sentencia siguiente, sin hacer nada.
- Tanto sentencia1 como sentencia2 son una única sentencia, pero donde puede ir una sentencia, puede ir todo un bloque de sentencias, sin más que encerrarlas entre llaves

Forma completa de la sentencia if con dos alternativas compuestas de varias sentencias

```
if (expresión-lógica){
   sentencia1;
   sentencia2;
   ...
   sentenciaN;
}else{
   sentenciaElse1;
   sentenciaElse2;
   ...
   sentenciaElseN;
}
```

Forma simple de la sentencia if con una única alternativa compuesta de varias sentencias.

```
if (expresión-lógica){
  sentencia1;
  sentencia2;
  ...
  sentenciaN;
}
```

Ahora Carmen está en condiciones de evitar que su programa aborte o muestre valores extraños cuando el número de días es cero. En concreto en el ejemplo que sigue, encontrarás que se han ejecutado cuatro if simples independientes, de forma que siempre habrá que comprobar las 4 condiciones.

☐ **Descarga el archivo zip SueldoMedioDiarioV3.java**

AUTOEVALUACION

Dada la siguiente sentencia, ¿cómo se evaluaría?:

```
if (expresión-lógica){
            sentencia1;
            sentencia2;
            ...
            sentenciaN;
 }else{
            sentenciaElse1;
            sentenciaElse2;
            ...
            sentenciaElseN;
  }
```

- a) Si al evaluar la expresión-lógica resulta verdadera, se ejecutará la sentencia1, y se continuará con la sentencia que haya después del if. Si por el contrario, la expresión-lógica resulta falsa, se saltará sin ejecutarse la sentencia1 y se ejecutará la sentencia2.
- b) Si al evaluar la expresión-lógica resulta verdadera, se ejecutará la sentencia1, la sentencia2,..sentenciaN y se continuará con la sentencia que haya después del if. Si por el contrario la expresión-lógica resulta falsa, se saltarán sin ejecutarse las sentencias anteriores al esle y se ejecutarán las siguientes: sentenciaElse1, sentenciaElse2... sentenciaElseN
- c) Se ejecuta la sentencia1 siempre que el valor no sea nulo.
- d) Todas las anteriores son falsas.

Comprobar

Unidad Didáctica IX

Condicional múltiple.

¿Piensas que serán frecuentes las situaciones en las que existan más de dos alternativas posibles? Seguro que sí. Y una forma de elegir la adecuada es haciendo preguntas, comparando de dos en dos las alternativas posibles, descartando una a una las distintas alternativas hasta que encontremos la que debemos realizar, en cuyo caso no habrá que realizar ninguna pregunta más.

Eso es lo que conseguimos con el **if múltiple**. Realmente no se trata de una sintaxis distinta. Simplemente se trata de un if que cuando se cumple la expresión lógica, la sentencia que ejecuta es a su vez un nuevo if (el if es a fin de cuentas una sentencia Java), que a su vez puede contener un nuevo if, y así hasta donde sea necesario.

```
if (expresion-logica-1)
    sentencia1;
else if (expresion-logica-2)
         sentencia2;
         ....
             else if (expresion-logica-N-1)
                      sentenciaN-1;
                 else
                      sentenciaN;
```

Si la expresión-logica-1 es cierta, se ejecuta la sentencia1 y se salta toda la parte else que abarca el resto de sentencias de la expresión.

Si por el contrario es falsa, habrá que seguir preguntando si se cumple expresión-lógica-2. Si es así se ejecutará sentencia2 y se saltarán el resto de sentencias. Si esta segunda expresión fuese falsa, habría que seguir preguntando y así sucesivamente hasta agotar todas las alternativas posibles.

Siguiendo con el ejemplo de Carmen, si en el caso anterior hemos usado 4 if independientes, si resulta que el número de días es mayor que cero, ya no resulta necesario comprobar las demás condiciones de los otros if, ya que no son condiciones independientes. Si es positivo, no puede ser negativo ni nulo.

Para tener en cuenta esto, y así mejorar la eficiencia del programa, vamos a modificar el ejemplo, usando if múltiples. Además, en el primer if introducimos las dos sentencias que hay que ejecutar cuando el número de días es positivo, en un bloque usando llaves.

☐ **Descarga el archivo zip con SueldoMedioDiarioV4.java**

AUTOEVALUACION

Dada la siguiente sentencia, ¿cómo se evaluaría?:

```
if (expresion-logica-1)
         sentencia1;
else if (expresion-logica-2)
         sentencia2;
         ....
                  else if (expresion-logica-N-1)
                           sentenciaN-1;
                      else
                           sentenciaN;
```

- a) Si expresión-logica-1 es cierta, se ejecuta sentencia1 y se salta toda la parte else que abarca el resto de sentencias de la expresión. Si por el contrario es falsa, habrá que seguir preguntando si se cumple expresión-lógica-2. Si es así se ejecutará sentencia2 y se saltarán el resto de sentencias. Si esta segunda expresión fuese falsa, habría que seguir preguntando y así sucesivamente hasta agotar todas las alternativas posibles.
- b) Si expresión-logica-1 es falsa, se ejecuta sentencia2 y se salta toda la parte else que abarca el resto de sentencias de la expresión. Si por el contrario es falsa, habrá que seguir preguntando si se cumple expresión-lógica-2. Si es así se ejecutará sentencia2 y se saltarán el resto de sentencias. Si esta segunda expresión fuese falsa, habría que seguir preguntando y así sucesivamente hasta agotar todas las alternativas posibles.
- c) Si expresión-logica-1 es falsa y expresión-logica-2 es falsa se ejecuta sentenciaN-1.
- d) Todas las anteriores son verdaderas.

Comprobar

Unidad Didáctica IX

Condicionales anidados.

¿Qué ocurriría si tras comprobar una condición tengo dos posibilidades, pero cada una de ellas me lleva a tener que decidir entre otras dos posibilidades? Además, esta "bifurcación en dos caminos" de cada nueva rama de una condición puede ser necesaria tantas veces como ramas o situaciones distintas tengamos que comprobar. ¿Cómo representaríamos esa situación?

La respuesta es mediante los if anidados, en los que cada rama de una sentencia if puede ser a su vez una sentencia if.

En realidad, los if múltiples estudiados anteriormente son un caso particular de los if anidados, es decir, cuando la sentencia controlada por cualquiera de los elementos del if es a su vez de tipo if:

```
if (condicion-1) {
   if (subcondicion-1-1)
       sentencia1-1;
   else
       sentencia1-2;
} else
   ....
```

Si **condicion-1** es verdadera, la ejecución pasa a la sentencia incluida en el if, que es un bloque cuyo contenido es también un **if**, controlado en este caso por **subcondicion-1-1**.

El siguiente trozo de código ilustra el uso de if anidados:

```
if (sueldoMensual > 2000)
   if (diasTrabajadosEnMes < 15)
       System.out.println("Mucho sueldo y poco trabajo");
   else
       System.out.println("Mucho sueldo y mucho trabajo");
else
   if (diasTrabajadosEnMes < 15)
       System.out.println("Poco sueldo y poco trabajo");
   else
       System.out.println("Poco sueldo y mucho trabajo");
```

Pero imaginemos que queremos un programa que sólo escriba un mensaje cuando se cobre más de 2000 Euros, habiendo trabajado menos de 15 días, o cuando se cobre menos de 2000 Euros habiendo trabajado más de 15 días, simplificando el código anterior, podríamos obtener la siguiente sentencia:

```
if (sueldoMensual > 2000)
   if (diasTrabajadosEnMes < 15)
       System.out.println("Mucho sueldo y poco trabajo");
else
   if (diasTrabajadosEnMes >= 15)
       System.out.println("Poco sueldo y mucho trabajo");
```

Pero la sentencia anterior no resuelve el problema de la forma que deseamos. Cuando las dos condiciones son ciertas, escribirá correctamente el mensaje, pero cuando **sueldoMensual > 2000 y diasTrabajadosEnMes >= 15,** escribirá un mensaje indicando que el sueldo es poco y el trabajo mucho, lo cual no es cierto, ya que el sueldo es mayor que 2000.

¿Por qué sucede esto?

Porque la **indentación es simplemente un aspecto que mejora la legibilidad del código, pero sin ningún efecto para el compilador**. Aparentemente el único else del ejemplo está asociado al primer if, ya que lo hemos colocado con la misma indentación, a la misma altura. Pero el margen no le indica al compilador a qué if va asociado ese else.

Java evita este tipo de malentendidos asociando siempre que haya dudas cada **`else`** al **`if`** más cercano, al último **`if`**. ¿Cómo interpreta realmente el compilador Java el ejemplo anterior?:

```
if (sueldoMensual > 2000) {
   if (diasTrabajadosEnMes < 15)
       System.out.println("Mucho sueldo y poco trabajo ");
   else {
       if (diasTrabajadosEnMes >= 15)
          System.out.println("Poco sueldo y mucho trabajo ");
   }
}
```

Las llaves no son necesarias realmente en este caso, ya que de todas formas es así como lo interpreta, pero sirven para aclarar los ámbitos de actuación de cada construcción.

Para conseguir el resultado deseado al escribir los mensajes, hubiese sido necesario escribir el código de la siguiente forma:

```
if (sueldoMensual > 2000) {
   if (diasTrabajadosEnMes < 15)
       System.out.println("Mucho sueldo y poco trabajo");
} else
   if (diasTrabajadosEnMes >= 15)
       System.out.println("Poco sueldo y mucho trabajo ");
```

En este caso, hemos encerrado el segundo if entre llaves y así no hay ninguna duda de hasta dónde llega la parte a ejecutar si la sentencia es verdadera. Al mismo tiempo, el else sólo puede ir asociado al primer if, ya que va fuera del ámbito del segundo if, al ir fuera de las llaves que contienen a éste.

AUTOEVALUACION

Dada la siguiente sentencia , ¿cómo se evaluaría?:

```
 if (condición-1) {
         if (subcondicion-1-1)
         sentencia1-1;
         else sentencia1-2;
} else
         ....
```

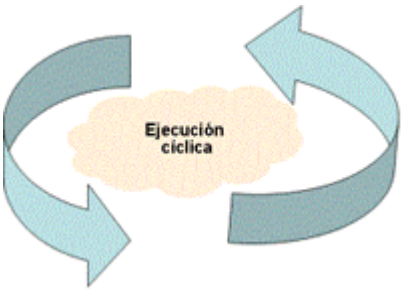
- a) Si condición-1 es verdadera, la ejecución pasa a la sentencia incluida en el if, que es un bloque cuyo contenido es también un if, controlado en este caso por subcondicion-1-1.
- b) Si condición-1 es verdadera y subcondicion-1-1 es verdadera se ejecutaría sentencia1-1.
- c) Si condición-1 es verdadera y subcondicion-1-1 es falsa se ejecutaría sentencia1-2.
- d) Todas las anteriores son correctas.

Comprobar

Sentencias y control de ejecución en java.

Unidad Didáctica IX

Ejecución cíclica.



¿Y si queremos ejecutar un grupo de sentencias más de una vez? ¿Tenemos que escribirlas repetidas veces?

Seguramente no. Lo más cómodo será disponer de alguna manera de indicarle al ordenador que repita la ejecución de esas sentencias, mientras que se cumpla una condición.

Por ejemplo, es posible que queramos calcular el sueldo medio diario para un conjunto de trabajadores, en vez de para uno sólo, por lo que deberemos repetir los pasos de leer el nombre del empleado, leer el número de horas que ha trabajado en el mes, y calcular su sueldo medio diario.

Esto es justamente lo que nos permiten hacer las sentencias de control de flujo repetitivas, o cíclicas o iterativas (son tres formas equivalentes de llamarlas).

En este apartado vamos a ver la sintaxis en Java de los tres tipos de sentencias cíclicas que hay en Java.

Unidad Didáctica IX

Condicional múltiple, o selección múltiple.

Hasta ahora hemos visto que con las sentencias **if** podemos ir tomando muchos caminos alternativos, pero siempre tendremos que elegir uno de entre dos posibles, y rechazar el otro. Pero en la vida diaria hay muchos casos en los que realmente se nos presentan no dos, si no muchas alternativas o caminos posibles, y sólo podremos tomar uno de ellos.

¿Cómo trata Java este tipo de situaciones?

Mediante el uso de una sentencia de alternativa o selección múltiple. La sentencia switch.

La sintaxis de esta sentencia es la siguiente:

```
switch (expresion-entera) {
   case valor1:
      sentencias1;
   case valor2:
      sentencias2;
   ....
   ....
   case valorN:
      sentenciasN;
   default:
      sentencias-default;
}
```

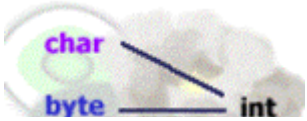
Veamos cómo debe interpretarse:

1. La expresión que controla el **`switch`** debe ser de tipo entero, es decir, debe devolver un valor de tipo entero.
2. Los tipos **`char`**, **`byte`** y **`short`** se convierten automáticamente a **`int`** (enteros) al operar sobre ellos, por lo que también pueden utilizarse en la condición de control de un **`switch`**.
3. Al contrario que otros lenguajes, no se puede utilizar como expresión de control nada que no dé como resultado un tipo entero.
4. Cada uno de los posibles caminos del condicional múltiple vendrá especificado por una cláusula **`case`** que se ejecutará cuando el valor asociado al **`case`** coincida con el valor obtenido al evaluar la expresión del **`switch`**.
5. Al contrario que otros lenguajes, las cláusulas case no pueden indicar condiciones, ni rangos de valores, ni listas de valores. Hay que indicar uno a uno, con su propio case, todos los valores posibles que queremos comprobar.
6. Es posible indicar un caso por defecto (**`default`**), que se activará si ninguno de los casos indicados mediante cláusulas case ha sido activado.
7. Es posible indicar un conjunto de sentencias para cada caso, formando un bloque, sin que ni siquiera sea necesario agruparlas entre llaves, ya que las cláusulas case delimitan sin ambigüedad sus ámbitos de actuación.
8. Se produce una ejecución denominada por caída hacia abajo. Es decir, una vez que un case se activa, la ejecución continúa por los siguientes case hasta que se encuentre una sentencia de interrupción:

```
break;
```

El efecto de break es salir del switch, y transferir el control a la siguiente instrucción que haya escrita.

Como todos esos detalles ya fueron comentados en la unidad 4, no vamos a volver a entrar en más detalles. De lo que se trata es de que veas el funcionamiento de esa estructura con un ejemplo típico.



Se trata de hacer la conversión de una nota entera, leída desde teclado, y que tendrá un valor entero del 0 al 10, a su expresión literal de Sobresaliente, Notable, Bien, etc. Ejecuta tú mismo el código, y comprueba sobre todo la llamada caída hacia abajo, que nos permite hacer lo mismo para un conjunto de posibles notas, sin tener que escribirlo más que una vez.

☐ **Descarga el archivo zip con Notas.java**

AUTOEVALUACION

Dada la siguiente sentencia, ¿cómo se evaluaría?:

```
switch (expresion-entera) {
       case valor1:
                sentencias1;
       case valor2:
                sentencias2;
     ....
       ....
       case valorN:
                sentenciasN;
       default:
                sentencias-default;
}
```

- ○ a) Si expresión-entera es igual a valor1 se ejecutarían sentencias1, si es igual a valor2 se ejecutarían sentencias2... si es igual a valorN se ejecutarían sentenciasN. Es decir, siempre se ejecuta una sentencia.
- ○ b) Si expresión-entera es igual a valor1 se ejecutarían sentencias1, sentencias2,...sentenciasN, sentencias-default. Es decir, se ejecutarían todas, al no incluir cláusulas break en cada bloque .
- ○ c) Si expresión-entera es distinta a valor1 se ejecutarían sentencias2, sentecia3... sentenciasN.
- ○ d) Todas las anteriores son ciertas.

Comprobar

Unidad Didáctica IX

Sentencia while.

¿Para qué casos es adecuada esta sentencia?

Para aquellos en los que no sabemos si las sentencias del bucle habrá que ejecutarlas o no.

Vamos a poner un ejemplo. Necesitamos hacer un cálculo para todos los números incluidos entre un máximo y un mínimo, que se leen desde teclado. En principio, un ciclo con una variable de control del bucle, a la que llamamos contador, que comience tomando el valor del mínimo, procese ese valor y le sume uno a la variable contador para procesar el siguiente valor, mientras que la variable sea menor o igual que el máximo, parece adecuado. Pero, ¿qué hacer si el valor introducido por teclado para el mínimo ya es mayor que el máximo? Lo lógico es que en este caso no se procese ningún valor, es decir, que las sentencias de procesamiento del bucle no se ejecuten ninguna vez. Y eso es justamente lo que hará el bucle usando la sentencia while si la condición es falsa cuando se llega al ciclo por primera vez.

```
/*Si el valor de minimo es mayor que maximo, la condición del while es  falsa, y el mét
procesar(contador) no se ejecuta ninguna vez*/

int contador = minimo;
while (contador<=maximo){
         procesar (contador);
         contador++;
}
```

Es decir, **que primero necesitamos comprobar si la condición es cierta, y si lo es se ejecutarán una vez, y se seguirán ejecutando mientras que la condición sea cierta**. **Si inicialmente la condición es falsa, el bucle no se ejecuta**. Por tanto es necesario que dentro de las sentencias del bucle haya algo que cambie el valor de verdad de la condición, para que en algún momento pueda dejar de ser cierta, y evitemos de esta forma quedar bloqueados en un bucle infinito.

La sintaxis de la sentencia while es como sigue:

```
while (expresión-lógica)
   sentencia;
```

Como siempre, si queremos que se ejecute un bloque de sentencias de forma cíclica en vez de una sola, basta con ponerlas todas entre llaves, en lugar de poner una única sentencia.

Es lo que se hace en el bucle del ejemplo siguiente, que procesa el número de trabajadores que le indiquemos por teclado, calculando su sueldo medio diario.

Una vez más te animo a que lo ejecutes y compruebes que funciona correctamente, asegurándote que entiendes el funcionamiento de la sentencia while.

☐ **Descarga el archivo zip con SueldoMedioDiarioCiclico.java**

Unidad Didáctica IX

Sentencia for.

Pero, ¿Se pueden facilitar las cosas si se conoce de antemano el número exacto de veces que hay que ejecutar el ciclo?

Ciertamente sí. Si sabemos cuántas veces se tiene que ejecutar el bucle, o si sabemos el rango de valores para los que se tiene que ejecutar, **el bucle for permite representar de forma más clara** el ciclo que hay que realizar.

La sintaxis de esta construcción es:

```
for (expresión-inicio; expresión-lógica; expresión-incremento)
      sentencia;
```

Un bucle for es de alguna manera una forma abreviada de representar un bucle **`while`**. Así, el mismo efecto se puede conseguir con estas sentencias equivalentes:

```
expr-inicio;
while (expr-lógica) {
   sentencia;
   expr-incremento;
}
```

Básicamente, un bucle for está controlado por tres expresiones:

- **La expresión de inicio**, sólo se ejecuta una vez antes de la ejecución del bucle propiamente dicho.
- **Si la expresión lógica de control es inicialmente falsa, el bucle no se ejecuta ninguna vez**.
- **La expresión lógica se evalúa en cada iteración**.
  - **Si el valor devuelto por esta expresión es verdadero (`true`), se ejecuta el bucle**, y la **`sentencia`**.
  - A continuación **se ejecuta la expresión de incremento, tras cada vuelta**.
  - De nuevo **se devuelve el control a la expresión lógica para decidir si se debe ejecutar de nuevo** el cuerpo del bucle.
  - Pero **si la expresión lógica devuelve un valor falso (`false`), entonces se interrumpe la ejecución del bucle**, y se continúa con la siguiente sentencia tras el **`for`**.
- **La expresión de incremento se ejecuta como paso intermedio en cada ciclo de iteración, tras la ejecución de la sentencia** que constituye el cuerpo del bucle y la evaluación de la expresión lógica o de control.
- **Tanto la expresión de inicio como la de incremento pueden estar formadas por varias sentencias**, en cuyo caso irán separadas por comas.
- **Cualquiera de las tres expresiones puede estar vacía**, aunque los puntos y coma de separación serán siempre necesarios.

El que las expresiones de inicio o incremento estén vacíos no supone nada ya que no se ejecuta nada (algo así como la sentencia nula), pero **si aparece en blanco la expresión lógica, entonces se considera que el valor por defecto es `true`**.

En el siguiente ejemplo puedes comprobar el funcionamiento de la sentencia for para calcular la suma de los números pares comprendidos entre 2 y 100, ambos incluidos.

☐ **Descarga el archivo zip con NumerosPares.java**

AUTOEVALUACION

Dadas las siguientes sentencias, ¿cuál será finalmente el valor de x?:

```
  x=3;
while(x!=0 && x>3 )
  x=x-1;
```

- a) No se ejecutaría el bucle ya que x es igual a 3, por lo que ese seguiría siendo su valor.
- b) 0
- c) 1
- d) Ninguna de las anteriores es correcta.

Comprobar

6.3.3. Sentencia do while

Unidad Didáctica IX

Sentencia do while.

Pero, ¿y si sabemos que con toda seguridad hay que ejecutar las sentencias del bucle al menos una vez, antes de comprobar la condición?

En ese caso, la estructura que nos proporciona la sentencia cíclica **`do while`** puede ser más adecuada.

El bucle **`do while`** es similar al **`while`**, pero con la diferencia de que la condición en lugar de ejecutarse al principio, se ejecuta al final del bloque de sentencias del bucle.

La sintaxis de esta sentencia es la que sigue:

```
do
   sentencia;
while (expresión-lógica);
```

La sentencia incluida en el bucle **`do while`** se ejecutará al principio. A continuación se evaluará la **`expresión-lógica`**. Si la expresión es falsa se finalizará la ejecución del bucle, y si es cierta se volverá a ejecutar el cuerpo (**`sentencia;`**)

Recuerda una vez más que donde se puede poner una sentencia también se puede poner un bloque de sentencias entre llaves.

En un bucle **`while`** puede que el cuerpo no se llegue a ejecutar nunca, lo cual sucede si al evaluarse la condición la primera vez ésta es falsa (lo que por ejemplo sucedía en la aplicación **`SueldoMedioDiarioCiclico.java`** cuando se indica un número de empleados negativo). Sin embargo, **en un bucle do while el cuerpo se ejecuta siempre al menos una vez, ya que la condición se evalúa una vez que se finaliza la ejecución del cuerpo.**

Como ejemplo, vamos a ver un programa que lee números de teclado hasta que se introduzca un número negativo, indicando el total de números leídos, su suma y su media. Puesto que al menos habrá que leer un número para saber si hay que continuar o no, incluir la lectura en un bucle **`do while`** parece adecuado para garantizarlo. Ejecuta y analiza el código hasta que estés seguro de que lo entiendes.

☐ **Descarga el archivo zip con MediaYSumaNumeros.java**

Unidad Didáctica IX

Sentencias de salto incondicional.

¿Pero no quedamos en la unidad 4 en que este tipo de sentencias no se debían usar, o que al menos se debían evitar, en la mayoría de los casos?

Desde luego que sí, y allí se explicaban los motivos, pero a fin de cuentas forman parte del lenguaje, y debemos conocer su sintaxis y su funcionamiento.

Incluso, bien usadas, pueden ajustarse bastante a los criterios de programación estructurada. Pero no insistiremos demasiado en ellas, ya que cualquier programa puede escribirse prescindiendo de ellas. Además, todas las que ofrece Java están pensadas para que no transfieran el control a cualquier sitio. Todas suponen saltos incondicionales hacia delante, y hacia una zona cercana del código, lo cual vimos en la unidad 4 que era compatible con la programación estructurada.

Unidad Didáctica IX

Sentencias break y continue.

La sentencia `break` ya ha aparecido asociada a la sentencia **`switch`**. También puede emplearse en el interior de sentencias cíclicas. En todos los casos, transfiere el control al final del ciclo **`while`**, **`do while`**, **`for`** o sentencia **`switch`** más interna en la que se encuentra.

Ejecutar esta sentencia sin incluirla en ninguna condición es altamente desaconsejable, y si va asociada a una condición, siempre podremos encontrar una alternativa en la que se pueda conseguir el mismo efecto sin usar break, por el procedimiento de incorporar la condición que controla el break a la condición de salida del bucle. Esta alternativa estructurada será preferible en la mayoría de los casos.

Por ejemplo, ante una construcción del tipo:

Salida del bucle for usando break (no estructurado)

```
while (condicion1) {
   sentencia1;
   for ( ; ; ) {
          sentencia2;
          if (condicion2)
                sentencia3;
          else
            break;
   }
   sentencia4;
}
```

Alternativa estructurada a al uso de break

```
while (condicion1) {
     sentencia1;
     for ( ;condicion2; ) {
        sentencia2;
        if (condicion2)
          sentencia3;
     }
     sentencia4;
 }
```

Usando **`for`**, una vez que se entra en el bucle **`for`**, como no se indica ninguna condición, éste sería en principio un bucle infinito, en el que se ejecutarían repetidamente **`sentencia2`** y la sentencia **`if`**. Pero si la condición2 que controla el **`if`** es falsa, se ejecutará la sentencia **`break`**;, que de acuerdo con la definición interrumpe el bucle más interno. En este caso, interrumpirá el **`for`**, transfiriendo el control a la **`sentencia4`**.

La sentencia `continue` también se usa para alterar el flujo normal de ejecución en las sentencias cíclicas **`while`** , **`do while`** y **`for`**. (**`continue`** no se puede usar con switch). En el caso de **`continue`**, lo que hace es devolver el control al bucle que se interrumpe, y que será el más interno, el que contiene la sentencia.

Veamos cómo se emplea:

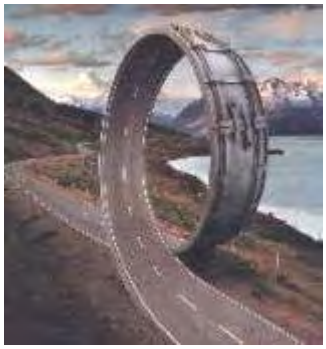
Finalización de la iteración usando continue (no estructurado)

```
while (condicion1) {
        sentencia1;
        while (condicion2){
                sentencia2;
                if (condicion3)
                        sentencia3;
                else
                        continue;
                sentencia4;
```

Alternativa estruct

```
while (condici
        sente
        while
                                    }
```

```
                }                                                    }
            }
```



Si en algún momento se llega a ejecutar la sentencia **`continue`**, lo que ocurre es que se transferiría el control al principio del bucle más interno que la contiene, **`while (condición2)`**, sin que llegara a ejecutarse la última sentencia del bucle, **`sentencia4`**.

Es como si se comprobara que algo va mal en la presente iteración del bucle, y se decidiera saltarse las sentencias que le quedan, para empezar a hacer otra nueva iteración desde el principio.

¿Y si el bucle que queremos interrumpir no es el más interno, sino un bucle más externo que está más "alejado" de la sentencia break o continue?

En ese caso, podemos indicar el lugar al que queremos transferir el control, mediante el uso de etiquetas.

Imagina que en el ejemplo anterior el **`continue`** debe devolver el control al principio del bucle correspondiente a while (**`condicion1`**), que no es el que contiene la sentencia **`continue`**, si no el más externo. Podemos conseguir esto de la siguiente forma:

```
bucleExterno:
   while (condicion1) {
      sentencia1;
      while (condicion2)
         sentencia2;
         if (condicion3)
            sentencia3;
         else
            continue bucleExterno;
         sentencia4;
      }
   }
```

Donde bucleExterno: es una etiqueta que colocamos en el lugar al que queremos transferir el control, justo delante del comienzo del bucle más externo.

Lo mismo puede hacerse con la sentencia break. No obstante el uso de este tipo de etiquetas, que permiten saltos del control del flujo hacia atrás y a zonas de código más remotas, son altamente desaconsejables, por enturbiar la claridad del código, y escasamente útiles.

Unidad Didáctica IX

Sentencia return.

¿Y si lo que queremos finalizar no es un bucle, sino la ejecución de un método antes de que termine? La sentencia return nos permite hacerlo.

En realidad la sentencia **`return`** se usa con una doble finalidad:

Termina la ejecución del método en el que se encuentre, transfiriendo el control al punto desde el que se hizo la llamada a ese método, continuando con la sentencia posterior.

Si va acompañado de una expresión de un determinado tipo, hace que el método devuelva un valor, es decir, que en el lugar donde se hizo la llamada al método, esa llamada se sustituya por el valor resultante de evaluar la expresión que acompaña al método.

Lo normal es que el return sea la última línea de un método, de forma que se cumpla el principio de programación estructurada de una entrada - una salida. Pero también puede usarse en cualquier punto de un método, y éste terminará en el mismo punto donde se ejecute ese return. Incluso podemos poner varios return en un mismo método. No obstante, tampoco suele ser aconsejable ni necesario, y siempre existe una alternativa estructurada en la que sólo aparezca un return al final de cada método.

Aunque volveremos a ver su funcionamiento cuando veamos con más detalle el uso de métodos y llamadas a métodos, podemos entender mejor el significado de return a través de alguno de los ejemplos vistos hasta ahora.

Concretamente el método **`leeNº ()`** de la clase ES que usamos para realizar la entrada por teclado para nuestros programas.

```
public static int leeNº(String mensaje) {
     int numero=0;
     boolean incorrecto=true;
     while(incorrecto) {
         System.out.println(mensaje);
         try {
            numero=Integer.parseInt(leeDeTeclado().trim());
            incorrecto=false;

         } catch(NumberFormatException e) {
            incorrecto=true;
            System.err.println("NO ES UN NÚMERO ENTERO VÁLIDO:
                                Vuelve a intentarlo.");
         }
      }
   return numero;
   }
```

Básicamente lo que hace es escribir un mensaje solicitando un número y formar una cadena con los caracteres introducidos por teclado hasta que se pulsa la tecla Intro. Esa cadena se intenta convertir en número entero, y si no es posible, vuelve a pedir otro número de nuevo. Una vez que se introduce un número correcto, el bucle while deja de ejecutarse y alcanzamos la sentencia return numero, que devuelve el número calculado.

Así, en la sentencia **`int cantidad = ES.leeNº("Introducir cantidad");`** al ejecutar el

método éste devuelve un número capturado desde teclado, y toda la llamada se sustituye por el valor devuelto, que se asigna a la variable cantidad.

AUTOEVALUACION

Señala la afirmación correcta. Respecto a la sentencia return, podemos afirmar que:

- a) Termina la ejecución del método en el que se encuentre, transfiriendo el control al punto desde el que se hizo la llamada a ese método, continuando con la sentencia posterior.
- b) Si va acompañado de una expresión de un determinado tipo, hace que el método devuelva un valor, es decir, que en el lugar donde se hizo la llamada al método, esa llamada se sustituye por el valor resultante de evaluar la expresión que acompaña al return.
- c) Puede usarse en cualquier punto de un método, y éste terminará en el mismo punto donde se ejecute ese return
- d) Todas las anteriores son correctas.

Comprobar

Respecto a la sentencia break, podemos afirmar que:

- a) Transfiere el control al final del ciclo while, do while, for o sentencia switch más interna en la que se encuentra.
- b) Ejecutar esta sentencia sin que dependa de ninguna condición es altamente desaconsejable.
- c) Si va asociada a una condición, siempre podremos encontrar una alternativa en la que se pueda conseguir el mismo efecto sin usar break, por el procedimiento de incorporar la condición que controla el break a la condición de salida del bucle.
- d) Todas las anteriores son correctas.

Comprobar

Unidad Didáctica IX

Descarga del Entorno Integrado de Desarrollo. (IDE)

__Carmen__ opina que __Víctor__ está ya preparado para practicar con el entorno de desarrollo. Conoce los tipos de sentencias, ha visto algunos programas y es capaz de diseñar y traducir algoritmos en Java. Ahora lo que necesita es practicar, experimentar con NetBeans y llevar a cabo el desarrollo de algunos ejercicios. __Carmen__ quiere que __Víctor__ no dependa de nadie al trabajar como programador y le explica cómo buscar, descargar e instalar esta aplicación. __Víctor__ le dice que sería más rápido si lo tuviera en disco, pero __Carmen__ no está totalmente de acuerdo, le dice que dadas las actuales líneas de comunicación de Internet es una tontería cargar con un paquete de discos con las herramientas, lo ideal es saber dónde encontrarlas e instalarlas desde cualquier equipo. Para ello __Carmen__ hace algo de trampa, tiene su propia página Web con una zona de uso restringido (a la que sólo ella puede acceder) en la que tiene los archivos de instalación de todas las herramientas que puede necesitar en un momento dado. Claro que para eso necesitas que el equipo desde el que intentas acceder tenga conexión a Internet, algo que no siempre se cumple, aunque cada vez menos.

Pero en esta ocasión no va a hacer uso de su página Web, lo que pretende es que __Víctor__ sea capaz de buscar la Web oficial de Sun (o a la del proyecto NetBeans), localizar el archivo adecuado para bajarlo y proceder a la instalación en su equipo. En lugar de darle el pescado, le está enseñando a pescar.

Hasta ahora has venido ejecutando y compilando los ejemplos con el Kit básico de desarrollo Java: el JDK, junto a un simple editor de texto plano, como el bloc de notas.

Habrás tenido ocasión de comprobar que el proceso es bastante laborioso y fatigoso, sobre todo si hay muchos errores en los programas. Y eso que todavía los programas son cortos, y por tanto fáciles de corregir.

Afortunadamente, existen Entornos Integrados de Desarrollo que nos permiten agilizar la tarea de escritura, compilación, depuración de errores y ejecución de nuestros programas. Estos entornos nos ofrecen todas las herramientas necesarias en una sola aplicación, que además resulta fácil de usar, intuitiva, y que hace gran parte de la tarea por nosotros.

En nuestro caso hemos optado, por las razones que comentábamos en la introducción de la unidad, por el NetBeans IDE, de código abierto, y proporcionado gratuitamente por Sun MicroSystem, la empresa desarrolladora de Java. Es un entorno en constante proceso de mejora y evolución, por lo que es posible que en breve aparezcan versiones más modernas que la actual. En cualquier caso, ésta cubre satisfactoriamente nuestras principales necesidades.

Unidad Didáctica IX

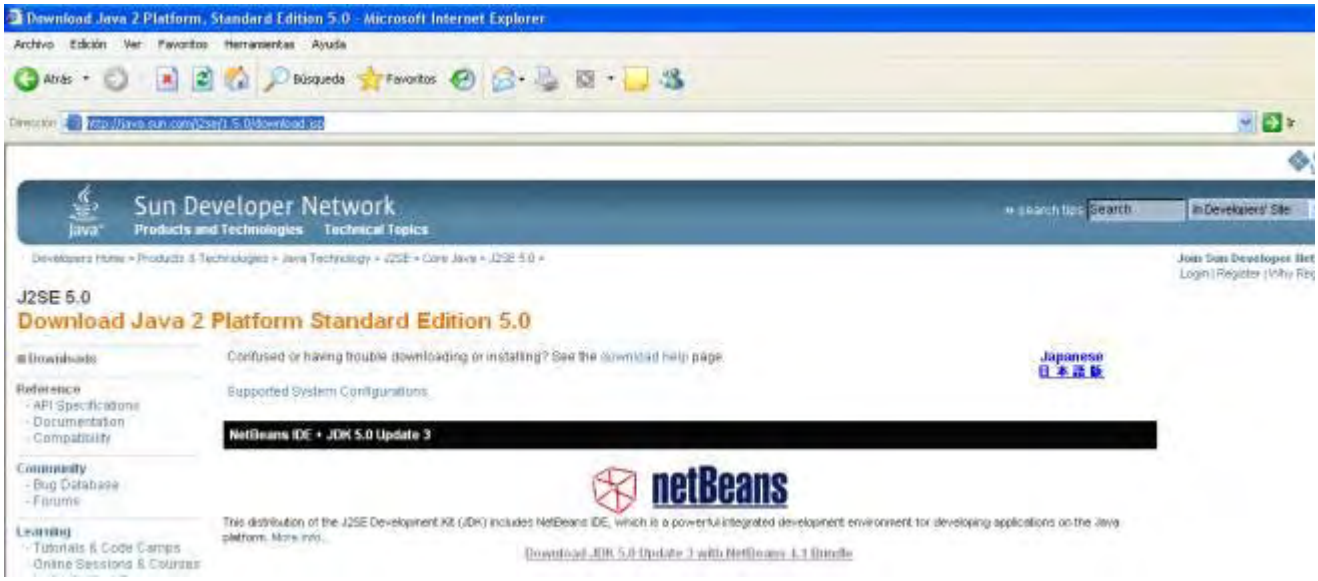
Descarga e Instalación del IDE junto al JDK de Java

DEMO: Vea como descargar J2SE.
Pulsa en el cursor para ver los distintos pasos para descargar J2SE.

ZONA DE DESCARGAS:

Existe la posibilidad, puesto que son suministrados por la misma empresa, de descargar el NetBeans IDE junto al más reciente JDK en un único paquete autoinstalable de la página de Sun.

Zona de Descarga del JDK más reciente.



ZONA DE DESCARGAS:

Pinchando en el enlace siguiente, de esa página, nos llevaría a la página de descargas del paquete. El texto del enlace es bastante descriptivo.

Descarga del paquete JDK 5.0 (Download JDK 5.0 Update 3 with NetBeans 4.1 Bundle)

Entre otras cosas adicionales, nos podemos descargar el JDK 5.0 Update 3 with NetBeans 4.1Bundle (el

jdk y el ide) pinchando en el icono 

En la pantalla de consentimiento con las condiciones de licencia, marcamos Accept y hacemos clic en Continue.

Seleccionamos nuestra plataforma y pulsamos sobre  para iniciar la descarga.

A partir de ese momento, sólo tenemos que seleccionar la carpeta de destino, y posteriormente ejecutar el fichero descargado para que se instalen tanto el Entorno como el JDK.

DEMO: Vea como instalar J2SE.

Pulsa en el cursor para ver los distintos pasos para instalar J2SE.

Unidad Didáctica IX

Descarga e instalación sólo del IDE.

Si has seguido el curso hasta ahora, seguramente ya habrás instalado el JDK con anterioridad, por lo que quizás sólo necesites instalar el NetBeans IDE

DEMO: Vea como descargar NetBeans IDE.
Pulsa en el cursor para ver los distintos pasos para descargar NetBeans IDE.

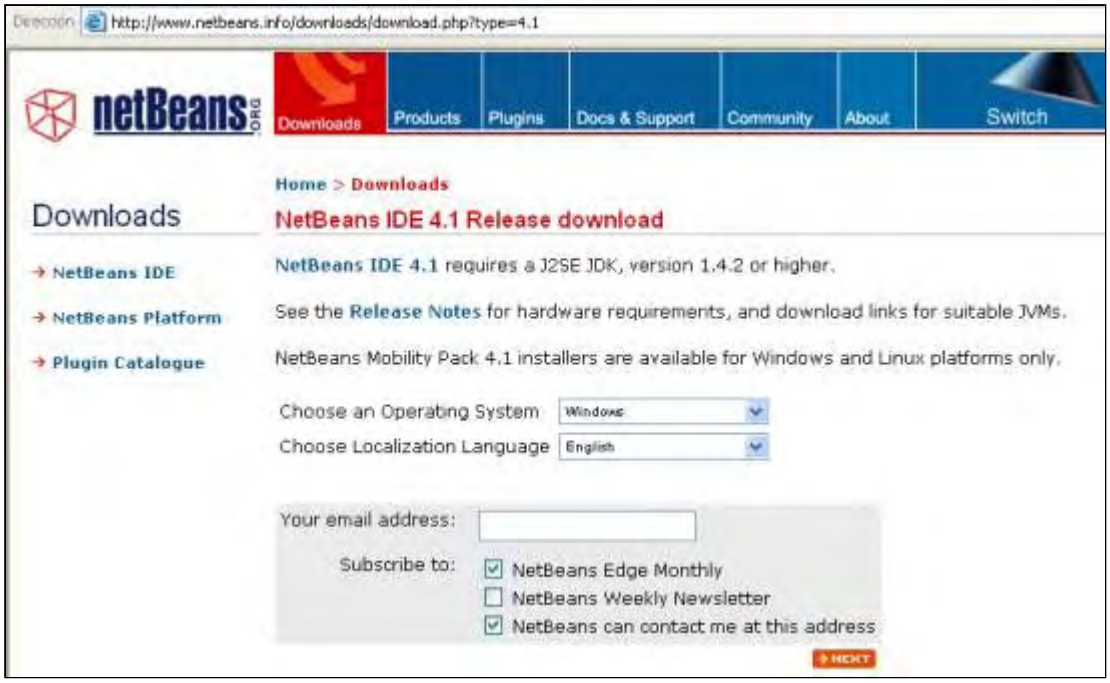
ZONA DE DESCARGAS:

También es posible descargarlo de forma independiente desde la página de Sun, pero existe la posibilidad de descargarlo directamente de la página del proyecto NetBeans, que es la siguiente:

Proyecto NetBeans

Entramos en la sección de descargas (Downloads) y elegimos descargar el NetBeans (la versión más reciente del momento)

Pinchamos en el enlace NetBeans IDE 4.1. download, elegimos el Sistema Operativo y el idioma (por el momento sólo inglés), indicamos nuestra dirección de correo, pulsamos NEXT en la pantalla siguiente y en la que le sigue, y comienza la descarga. Ya sólo es cuestión de indicar donde queremos guardar el fichero descargado, y de ejecutarlo para que se instale el IDE.



DEMO: Vea como instalar NetBeans IDE.

Pulsa en el cursor para ver los distintos pasos para instalar NetBeans IDE.

Unidad Didáctica IX

Escritura y ejecución de programas usando el IDE.

*La localización de la aplicación, la descarga y la instalación, no le han supuesto obstáculo alguno a **Víctor**, al que todo esto no le ha llevado más de quince minutos, la mayor parte de ellos se ha ido en la descarga del archivo de instalación (y con una buena línea ADSL). Ahora sí que está dispuesto para empezar a escribir los programas (aunque él prefiere copiar y pegar el código que ya tiene escrito) para después ejecutarlos y comprobar su funcionamiento. **María** le dice que a partir de ahora ha comenzado una nueva etapa en su vida, desde este momento tiene todas las posibilidades para ser un programador de ordenadores capaz de elaborar aplicaciones que harán más sencillas muchas tareas para sus usuarios.*

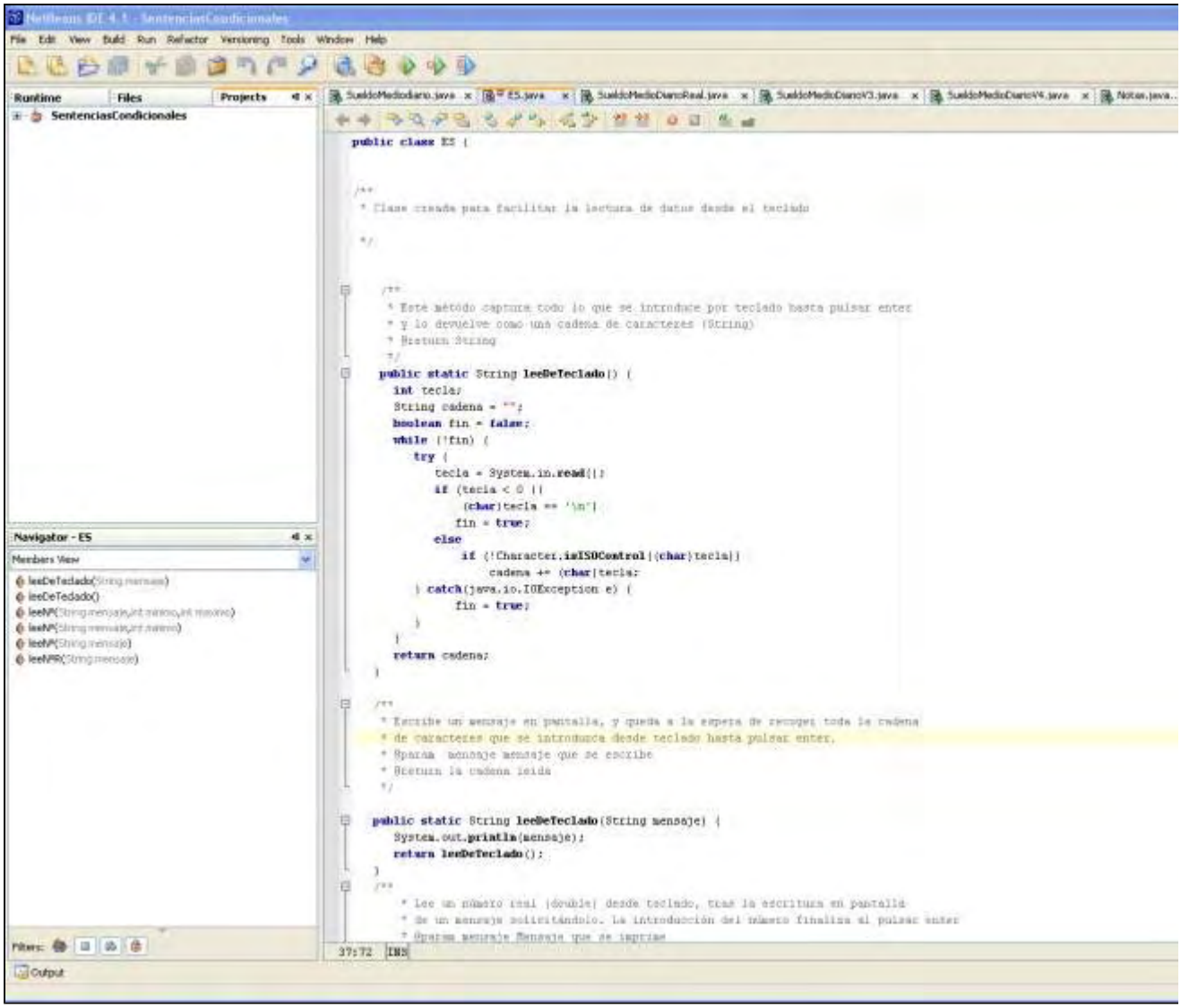
¿Ansioso/a de empezar a usar NetBeans? ¿Cansado/a de usar el bloc de notas? Llegó tu momento.

Como habrás imaginado, estaría bien que hicieras los ejercicios de la tarea de esta unidad usando NetBeans. En esta unidad sólo vamos a introducir los pasos básicos para que puedas escribir, compilar y depurar ejercicios sencillos del estilo de los que te pedimos en estas primeras unidades. Posteriormente dedicaremos una unidad a mejorar el conocimiento del entorno, y algunas de sus características más útiles.

En esta unidad se trata sólo de que veas a través de un ejemplo cómo se escribe, compila, y ejecuta un programa usando NetBeans.

Una vez instalado, en el escritorio debe aparecer el icono de NetBeans que al hacer doble clic sobre él nos abre la aplicación.

El aspecto de la aplicación es más o menos como el que sigue:



Y a partir de aquí, todo el proceso lo podrás seguir a través de las demostraciones que te vamos a presentar.

DEMO: Vea como trabajar con NetBeans IDE.
Pulsa en el cursor para ver los distintos pasos para ver NetBeans IDE en funciona.