Tema 3. Estructuras de programación

Contenido

Objetivos

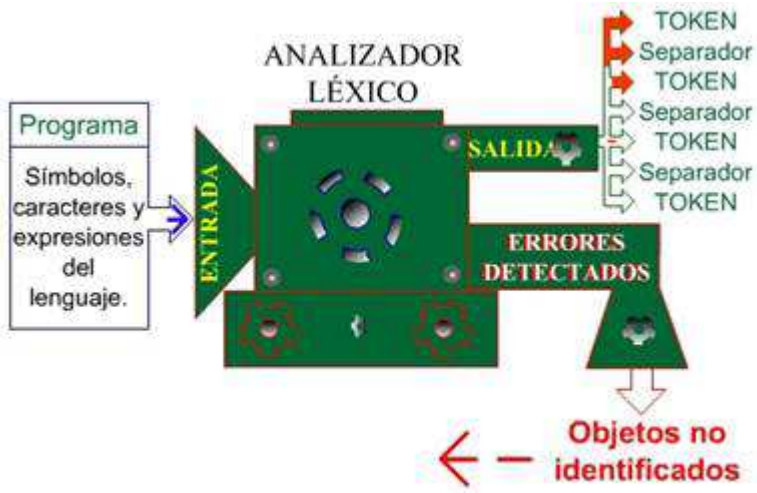
- Entender la diferencia entre léxico, sintaxis y semántica.
- Identificar los componentes léxicos de Java
- Construir expresiones simples
- Evaluar expresiones simples
- Utilizar métodos de Java en expresiones
- Entender el concepto de control de flujo respecto a la selección
- Comprender el cómo funciona el control de flujo anidado
- Entender la diferencia entre una estructura condicional y una serie de estructuras condicionales
- Saber cómo revisar el código y por qué
- Comprender el propósito de realizar la traza de un programa

1 Introducción

Anteriormente hemos visto que para poder ejecutar un programa era necesario traducirlo a código máquina, y que una de las formas más habituales de hacerlo era mediante el uso de compiladores.

Un lenguaje de programación es un conjunto de reglas, símbolos y palabras especiales que se utilizan para construir el programa. Por eso habrá que analizar el texto de dicho programa buscando e identificando elementos que tengan un significado especial. Uno de estos elementos serán las **instrucciones**, pero de la misma forma que cuando hablamos usamos frases u oraciones, que se componen de sintagmas, las instrucciones de un lenguaje de programación contienen otros elementos, que llamamos **expresiones**. Y al igual que el predicado de una oración necesita casi siempre de un verbo, las expresiones se suelen construir usando operadores. El proceso viene a ser el siguiente, de forma aproximada:

- **Cuando el compilador empieza a analizar el texto del programa escrito en lenguaje de alto nivel, lo que recibe es una secuencia de caracteres pertenecientes al alfabeto de ese lenguaje** (Unicode, en el caso de Java, ASCII, en otros lenguajes).
- **La primera tarea que debe realizar es aislar los elementos que tienen significado por sí mismos**, lo que usando la analogía con el lenguaje natural serían las palabras que forman ese texto (ese programa, en nuestro caso). A esa fase se le llama **análisis léxico**.
- **Para esto se sirve de unos elementos especiales que denominamos separadores.** Un **separador** no es más que un carácter, un símbolo o una palabra escrita en el texto del programa que le indica al compilador dónde termina una palabra con significado propio para el lenguaje y dónde empieza la siguiente. Cada una de las "palabras" que reconoce el compilador recibe en programación el nombre de **token**.



Simulación de lo que puede ser un analizador léxico.

Tipos de separadores

Los elementos que se usan como separadores se parecen a los que usamos en el lenguaje escrito para separar unas palabras de otras o para terminar una frase. Lo que se ha hecho en los lenguajes de programación es imitar, una vez más, al lenguaje natural.

En la mayoría de los lenguajes, los elementos que se consideran como separadores son los siguientes:

- **Espacio en blanco**. El espacio en blanco es un carácter más del alfabeto del lenguaje. Es el separador típico, y al igual que hacemos en el castellano escrito usándolo para separar palabras distintas, en programación se usa para separar tokens distintos.
- **Salto de línea**. El carácter de cambio de línea, o enter, o intro o new line, o return (todos esos nombres recibe) se usa con el mismo significado que el espacio en blanco. La única diferencia es la visual a la hora de leer el texto escrito. No obstante, podríamos escribir un programa entero en una sola línea (sin usar return), o con cada token en una línea distinta (sin usar espacios en blanco) o con cualquier mezcla de ambas situaciones, y el resultado para el compilador sería exactamente el mismo, ya que lo que hace realmente es sustituir cualquier separador o cualquier secuencia de separadores consecutivos por un único espacio en blanco.
- **Tabulador**. Al igual que return, se trata de otro posible separador equivalente al espacio en blanco.
- **Comentario**. Un **comentario** cumple asímismo las funciones de separador. Los comentarios tienen la función de documentación del código fuente, pero desde el punto de vista léxico del lenguaje, no son más que separadores. No representan ninguna acción a realizar, y por tanto se pueden suprimir al realizar la traducción a código máquina. Los lenguajes disponen de algún modo de indicar que parte del texto es un comentario, normalmente encerrándolo entre algunos caracteres especiales. Por ejemplo, en Java hay varios modos:

```
/* comentario */

/**
  comentario
   */

// comentario hasta fin de línea
```

En los 2 primeros ejemplos, la palabra comentario no tiene por qué estar compuesta de una sola línea.

El segundo tipo de comentario recibe el nombre de comentario javadoc. Son comentarios especiales, ya que el lenguaje Java es capaz de generar a partir de ellos documentación sobre la aplicación en formato html, formato web, que directamente se pueda publicar en Internet. No obstante, para el compilador, son comentarios similares a los del primer tipo, que se ignoran.

Tipos de Tokens

En el lenguaje natural, todas las palabras no son iguales a la hora de formar una frase con sentido, bien construida. El orden de las palabras es importante. No interpretamos igual los verbos que los artículos, ni podemos usarlos en el orden que queramos.

Lo mismo ocurre en los lenguajes de programación. En el proceso de análisis léxico, se han detectado los tokens que forman el programa. Pero existen distintos tipos de tokens que veremos en los siguientes apartados.

2 Palabras reservadas

Son aquellos tokens que tienen asignada una función específica en el lenguaje y que los programadores no pueden utilizar más que para lo que el lenguaje establece.

Por ejemplo, en Java, la palabra **int** sólo puede usarse para declarar una variable de ese tipo entero. Gran parte del aprendizaje de un lenguaje consiste en conocer el significado y la función de las palabras reservadas de ese lenguaje. La lista de palabras reservadas de un lenguaje, en contra de lo que pueda parecer, no es excesivamente larga. En el caso de Java, está formada por menos de 50 palabras:

abstract	double	int	super
boolean	else	interface	switch
break	extends	long	synchronized
byte	final	native	this
case	finally	new	throw
catch	float	package	throws
char	for	private	transient
class	goto	protected	try
const	if	public	void
continue	implements	return	volatile
default	import	short	while
do	instanceof	static	

En el tema anterior hemos visto qué son los literales, identificadores, operadores, la precedencia de los operadores y una noción de lo que son las expresiones.

En este tema vamos a profundizar un poco más en el concepto de **exprexión**.

3 Expresiones

Hemos visto en el apartado anterior cómo consigue el compilador aislar las distintas palabras (los tokens), y también que puede haber distintos tipos de tokens. La sintaxis es el conjunto de reglas que

definen exactamente qué combinaciones de letras, números y otros símbolos pueden utilizarse en el lenguaje de programación. La sintaxis de un lenguaje de programación no deja lugar a la ambigüedad porque el ordenador no puede pensar; no sabe qué queremos decir.

Los literales, junto con los identificadores y los operadores se pueden combinar para formar unidades de mayor significado, que denominamos **expresiones**. De esta forma la expresión podrá evaluarse, (evaluar una expresión consiste en calcularla, sustituir cada elemento por su valor, y hacer las operaciones indicadas) dando como resultado un valor determinado. La expresión tendrá asociado un tipo, y ese tipo será el del valor resultante de su evaluación.

En programación podríamos pensar en la expresión Java:

```
n + 1
```

Por sí misma aún no indica nada, salvo una operación que genera un resultado con el que no sabemos qué hacer, pero tiene más sentido que cada una de las partes por separado. Sin embargo si le decimos que el resultado lo guarde en una variable llamada **suma**, ya si tenemos una acción completa, una sentencia o instrucción.

```
suma = n + 1;
```

Las expresiones, combinadas con algunas palabras reservadas y a veces por sí mismas, forman **sentencias** o **instrucciones**.

Además de las expresiones, en las que de alguna forma se lleva a cabo la manipulación aritmética, lógica y relacional de la información, son necesarios elementos adicionales del lenguaje que controlen el orden de ejecución de las sentencias o instrucciones.

3.1 Definición de expresión:

1. **Todo literal de un determinado tipo es también una expresión de ese mismo tipo**.
2. **Todo identificador de un determinado tipo (una variable o constante) es también una expresión de ese mismo tipo.**
3. **Dado un operador *n*-ario que requiere *n* operandos (expresiones) de los tipos t_1, t_2, ..., t_n, y dados *n* operandos de los tipos correspondientes, el resultado es una expresión del tipo generado por el operador.**

La noción de expresión es bastante potente debido a la **recursividad** de la definición.

3.2 Expresiones matemáticas o aritméticas

Acabamos de ver la definición de expresión, y hemos visto que estaba ligada al concepto de tipo: todas las expresiones son de un tipo, están asociadas a un tipo de datos. Por lo tanto hay tantos tipos de expresiones como de datos.

Las expresiones matemáticas o aritméticas se forman mediante el uso de los operadores matemáticos o aritméticos asociados a los tipos numéricos básicos, junto a operadores de asignación. Generarán como resultado un valor numérico de alguno de los tipos definidos en el lenguaje.

Es posible que en una expresión matemática participen literales, identificadores (variables o constantes) o subexpresiones de distintos tipos numéricos, si bien es cierto que para operar entre ellos, habrá que hacer ciertas conversiones de tipo (también denominadas **casting**) para hacer posibles las operaciones. Esas conversiones de tipo pueden ser implícitas -automáticas- (las realiza por su cuenta el compilador) o explícitas (debe obligar a realizarlas el programador indicándolo expresamente).

Por ejemplo:

```
double a = 3 / 2.0;
```

- El operador / es la división, tanto entera como real.
- Si alguno de los dos operandos es de tipo real, será la división real la que se efectúe, es decir, obteniendo decimales.
- El compilador pasa por su cuenta el literal entero de tipo **int 3** a **3.0**, que es un literal real de tipo double, y realiza la operación real, obteniendo como resultado un double.
- **Eso es un casting o conversión implícita, también llamada promoción automática de tipos.**

Otro ejemplo:

```
byte var = 100;
var = (byte) (var + var);
```

- El tipo byte no tiene operadores propios sino que usa los del tipo int.
- Por tanto para hacer la suma (var + var) debe convertir el valor de var a int de forma implícita, para poder operar.
- Pero si quiero guardar el resultado de la operación, que es de tipo int y se almacena usando 32 bits, deberé forzar la conversión a byte (almacenado en 8 bits).
- En Java eso se consigue indicando entre paréntesis, y delante de la expresión, el tipo al que hay que convertir el resultado de la misma.
- **Eso es una conversión explícita (o casting explícito).**

Como se ha indicado en el ejemplo, los valores de tipo entero toman el tipo de datos **int**, mientras que los de punto flotante utilizan el tipo **double**. Esto afecta también a los literales por lo que, si deseamos que un literal en punto flotante sea almacenada en un elemento de tipo ***float*** en lugar de un ***double*** deberemos añadir la letra **F** (en mayúsculas o minúsculas) al final de dicho número.

Cuando realizamos una conversión de tipo (sea o no automática) debemos tener en cuenta de qué tipo de dato a qué otro tipo estamos convirtiendo. Por ejemplo, si almacenamos el resultado de una expresión de tipo ***int*** en una variable de tipo ***double*** no existirá pérdida de información: un número de 32 bit puede ser representado por completo en la mantisa de 53 bit. Cuando la conversión no representa una pérdida de información, se dice que es una **conversión segura** (***widening conversión***) –también llamada conversión sin pérdida o de ampliación-. Sin embargo, almacenar el resultado de una expresión de tipo ***float*** en un variable de tipo ***int*** puede llevar aparejada una pérdida de información porque la parte fraccional se trunca. Java se refiere a la misma como **conversión no segura** (***narrowing conversion***) –también llamada conversión con pérdida-.

En algunos casos, las conversiones no seguras pueden no funcionar, siendo el compilador el que presente un mensaje de error –aunque algunos errores se producen en tiempo de ejecución-. En esas situaciones será necesario realizar un conversión explicita. No obstante, para hacer nuestro código tan claro (y libre de errores) como sea posible, deberemos especificar las conversiones de forma explícita. Con el uso

de la conversión explícita resulta completamente claro para el programado y para aquellos que lean el código que la mezcla de tipos de datos es intencionada, no inadvertida.

3.3 Expresiones lógicas o booleanas

En los lenguajes de programación, las afirmaciones toman la forma de expresiones lógicas o booleanas. De la misma manera que una expresión aritmética está compuesta de valores numéricos y operaciones, una expresión lógica se compone de valores lógicos y operaciones. Toda expresión lógica tiene uno de los dos valores booleanos: **true** (verdadero) o **false** (falso).

Una expresión lógica puede ser una variable o constante booleana, es decir, que ha sido declarada como de tipo **boolean**.

Otra forma de asignar un valor a una variable booleana es haciéndola igual al resultado de comparar dos expresiones con un operador relacional.

Por ejemplo:

```
Boolean menorQueCero = (i < 0);
```

- Comparamos i y 0 con el operador "menor que".
- Mediante dicha comparación afirmamos que existe la relación "menor que" entre ellos.
- Si dicha relación existe la afirmación es cierta (true); si no, es falsa (false).
- Asignamos a la variable menorQueCero el valor obtenido.

Por supuesto, tenemos que tener cuidado con los tipos de datos de las cosas que comparamos. Lo más seguro es comparar siempre elementos del mismo tipo de datos: ***char*** con ***char***, ***int*** con ***int***, etc. Si mezclamos tipos de datos en la comparación, se producirá una conversión implícita, al igual que sucede con las expresiones aritméticas; también aquí es deseable utilizar conversión explícita para que se sepa que es intencionado.

Si se intenta comparar un valor booleano con uno numérico se producirá un error. Los valores booleanos sólo pueden ser convertidos a valores de tipo cadena; habitualmente, esto sólo se hace cuando una variable booleana es concatenada con una de tipo ***String***, de manera que su valor es transformado a "true" o "false" automáticamente. Los objetos de tipo ***String*** no pueden ser comparados en la forma habitual; trataremos esto en el tema 4.

Los operadores lógicos toman valores booleanos como operandos. Combinando operadores lógicos y relacionales podemos crear afirmaciones más complejas.

3.3.1 Evaluación en cortocircuito

Algunos lenguajes de programación utilizan la evaluación completa cuando procesan expresiones como:

```
i == 1 && j < 2
```

En ese caso, el ordenador evalúa primero ambas subexpresiones relacionales (i==1 y j<2), para aplicar después el operador && y determinar el resultado final.

Sin embargo, Java utiliza la denominada **evaluación en cortocircuito** (o condicional) de expresiones lógicas. En este caso, la evaluación se realiza de izquierda a derecha y la evaluación se detiene tan pronto como sea posible –tan pronto como se conoce el valor de toda la expresión booleana-.

¿Cómo puede el ordenador saber si la expresión es verdadera o falsa sin evaluar toda la expresión? Si observamos la expresión de referencia y suponemos que el valor de i es 74, la primera subexpresión es falsa; ello conlleva que el resultado de la operación con && será falso, independientemente del valor de j y el resultado de dicha subexpresión. El ordenador detendrá la evaluación y devolverá como resultado el valor

false. En el caso del operador || la evaluación se detendrá tan pronto como se encuentre una subexpresión que obtiene valor ***true***.

4 Funciones

¿Qué hacer si queremos realizar algún tipo de operación que no está definida por los operadores básicos del lenguaje o que nadie había necesitado antes?

La mejor solución es un mecanismo flexible para que se puedan proporcionar una serie de operaciones que se puedan usar además de los operadores de los tipos básicos, o incluso permitir la definición de nuevos operadores por el programador que se adapten a sus necesidades.

La mayoría de los lenguajes definen toda una serie de funciones que permiten realizar múltiples operaciones sobre los datos, obteniendo datos de los más diversos tipos.

Por ejemplo, es normal que los lenguajes incorporen toda una batería de funciones matemáticas, tales como raíces, valor absoluto, redondeo, funciones trigonométricas, logaritmos, potenciación, etc. que son utilizables con datos de tipo numérico y que pueden formar parte de cualquier expresión matemática, ya que la llamada desde el programa a estas funciones se sustituye por el valor que devuelven (el valor que calculan).

Por ejemplo, en Java, todas esas funciones matemáticas se encuentran disponibles en una clase llamada Math, y se pueden formar expresiones matemáticas como la que sigue, en la que se le asigna a la variable **a** el valor absoluto de -3, o sea 3:

```
int a = Math.abs(-3);
```

Pero el concepto de función en un lenguaje es mucho más general que el de función matemática, y realmente se extiende a cualquier procedimiento o método que devuelve un dato, de cualquier tipo.

Las estudiaremos con más profundidad más adelante.

5 Instrucciones o sentencias

Hasta ahora hemos estado viendo cómo a partir de los caracteres el compilador formaba palabras (tokens) y cómo a partir de los tokens se formaban las expresiones.

En el lenguaje de programación, tendremos que formar las **sentencias** o **instrucciones** a partir de las distintas expresiones que hemos visto hasta ahora.

Las **sentencias** en un lenguaje de programación representan acciones completas a realizar por el programa. Algunas de las sentencias tendrán como misión controlar el orden de ejecución de las demás, y serán las llamadas sentencias de control del flujo de ejecución. Todo programa no es más que un conjunto de sentencias o instrucciones.

Debemos tener en cuenta que:

- **Sólo las expresiones de asignación forman sentencias por sí mismas, sin más que añadirles algún carácter o token de finalización de sentencia. Ese carácter en Java es el punto y coma.**
- **Todas las sentencias deberán terminar en punto y coma**.
- **Serán sentencias de asignación las obtenidas por el uso del operador de asignación básico o por cualquiera de los operadores combinados de asignación**.
- **También se consideran sentencias de asignación las obtenidas mediante los operadores de incremento o decremento, ya que llevan implícita una asignación.**

Ejemplos de sentencias de asignación:

```
a = 3;
a += 3; // a = a + 3
a++;    // a = a + 1
```

¡Advertencia! Es muy habitual confundir el operador de asignación (=) con el de comparación (==). Estos dos operadores tienen efectos muy diferentes en el código. Algunos programadores denominan al operador relacional "igual-igual" para recordarse a ellos mismos la diferencia.

Existen tres tipos básicos de sentencias:

- **Sentencias de expresión**. **Están formadas por una expresión seguida del símbolo de terminación** (el punto y coma, en Java). En los párrafos anteriores hemos mencionado las sentencias de asignación, que constituyen el principal grupo de sentencias de expresión.
- **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, 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.

5.1 Sentencias de expresión

Hemos comentado que las sentencias de expresión se forman añadiendo punto y coma a una expresión. Pero ¿cualquier expresión será válida? Hemos visto que cualquier literal o cualquier identificador son en sí mismo expresiones, pero las siguientes expresiones seguidas de punto y coma no tienen sentido:

```
3.0;
numero;
"hola";
4;
```

Estas expresiones por sí solas no constituyen una "acción", algo que deba hacer el ordenador. Añadir punto y coma sin más a cualquier expresión no aumenta el significado de ésta.

No todas las expresiones pueden formar sentencias de expresión. O más concretamente, no todos los operadores dan lugar a expresiones susceptibles de formar sentencias.

5.1.1 Tipos de expresiones que generan sentencias de expresión

Una vez visto que no todas las expresiones pueden generar sentencias sin más que añadir punto y coma, vamos a ver cuáles son las que sí las generan, las expresiones que ya tienen significado por sí mismas:

- **Expresiones de asignación**: es decir, expresiones cuyo operador raíz es bien el igual (=) o un igual compuesto (op=). Ya nos hemos referido a ellas en el apartado anterior, a modo de introducción al hablar del concepto de sentencia. Una de las acciones más simples que le podemos dar a un ordenador es indicarle que almacene un valor en una variable. Por ejemplo:

```
a = 3;
a += 3;
```

- **Expresiones de incremento y decremento**: se trata de las expresiones formadas por las formas prefijas o postfijas de los operadores de incremento (++) o decremento (--). Estas expresiones en el fondo pueden considerarse como expresiones de asignación, por llevar una asignación implícita.

```
a++;
```

- **Llamadas a métodos o procedimientos**: Un método es una función de una clase en Java que realiza una cierta operación sobre dicha clase. Por ejemplo, abs() es un método de la clase Math para calcular el valor absoluto de un número. La llamada a un método, pasando los correspondientes **argumentos**, es así mismo una expresión. Y dicha expresión, seguida del símbolo de terminación, constituye una sentencia del tipo denominado sentencias de expresión.

  Si por ejemplo, introducimos todas las operaciones necesarias para sacar el listado de los trabajadores de una empresa en un método llamado listarPorNombre(), al que se le pasa como argumento un String, que será el nombre a listar. Cuando queramos usarlo, tendremos que hacer una llamada a este método indicando el nombre concreto que queremos listar. Una llamada tendrá la forma:

```
listarPorNombre("Juan");
```

  Tiene un significado claro. Queremos que se hagan todas las operaciones necesarias para que se muestre el listado de todos los trabajadores cuyo nombre sea "Juan".

- **Expresiones de creación de objetos**: La utilización del operador **new** para crear objetos genera expresiones de creación de objetos. Estas expresiones, seguidas del símbolo de terminación, dan lugar también a sentencias.

  En nuestro ejemplo tenemos claramente la necesidad de definir lo que para nosotros va a ser un trabajador. En el tema 2 ya vimos de forma muy breve lo que eran las clases y los objetos. En nuestra aplicación, tendremos que definir lo que es un trabajador en una clase llamada Trabajador. Hay que definir qué es un trabajador, y qué vamos a poder hacer con él. Posteriormente, cada vez que se dé de alta a un nuevo trabajador en la empresa, la aplicación deberá crear un nuevo objeto de tipo Trabajador, una nueva instancia de la clase Trabajador. Eso conlleva que algún programa se debe encargar de buscar espacio libre en memoria donde alojar los datos correspondientes a ese **nuevo objeto**. Esto es lo que hace el operador new. Veamos un ejemplo.

```
Trabajador  t1 = new Trabajador("Juan Sanchez Martinez");
```

  Estamos indicándole al compilador que busque en memoria espacio libre para alojar un nuevo objeto de tipo Trabajador, que el Trabajador lo tiene que crear con el nombre " Juan Sanchez Martinez", y alojarlo en esa posición, a la que nos vamos a referir mediante la variable **t1.**

Tanto las llamadas a métodos como la creación de objetos se estudiarán detenidamente más adelante.

5.2 Sentencias de declaración

Hemos visto que las sentencias tienen sentido completo, indican alguna acción a realizar por el programa. Hemos visto también que el segundo tipo de sentencias lo constituyen las sentencias de declaración. Se trata sólo de indicar el tipo junto al nombre de una variable. Al declarar una variable, estamos dando una orden concreta al programa.

Sería algo como lo que sigue:

- "Comprueba el tipo de la variable para ver qué tamaño ocupan en memoria los valores que va a almacenar."
- "Busca una zona de memoria que esté libre y donde quepan los valores de ese tipo."
- "Asocia esa zona de memoria con el nombre o identificador de la variable."

- "A partir de aquí, sustituye cualquier aparición del nombre de esa variable por la zona de memoria que tiene asociada."

5.2.1 Posibles formas de declarar variables

Las sentencias de declaración son siempre numerosas en los programas, y si conseguimos reducir y abreviar el código necesario para escribirlas, habremos ayudado a simplificar nuestro programa. Por eso existen varias opciones que facilitan la labor de declarar e inicializar las variables de un programa:

- **Una sentencia de declaración puede declarar simultáneamente varias variables si son del mismo tipo:**

```
int var1,var2,var3,var4;
```

  En esta sentencia se han declarado cuatro variables de tipo int. La sentencia anterior es en todo equivalente a:

```
int var1;
int var2;
int var3;
int var4;
```

- **Una sentencia de declaración permite asignar además del tipo un valor inicial a una variable:**

```
int var = 25;
```

  Lo que se coloca en el lado derecho puede ser cualquier expresión del lenguaje, que devuelva un valor del tipo correspondiente a la declaración. De esta forma, estas sentencias de declaración e inicio se convierten en una mezcla donde interviene tanto una sentencia de declaración como una sentencia de expresión, en concreto, de asignación.

- **Es posible mezclar la capacidad de inicialización de la sentencia de declaración con la posibilidad de declaración de múltiples variables:**

```
int v1, v2 = 2, v3;
```

  Pero en este caso es importante tener en cuenta que aunque se hayan declarado tres variables de tipo **int**, sólo **v2** ha sido inicializada. Las dos variables restantes han sido declaradas, pero no se les ha asignado ningún valor inicial.

No todos los lenguajes obligan a declarar las variables antes de ser usadas. Pero el hecho de no saber cuál va a ser el tipo de una variable hace que no se sepa cuanto tamaño va a ocupar, e impide al compilador reservar en memoria el espacio necesario para la misma, con lo que no se puede gestionar de forma eficiente la memoria.

Por ello, la mayoría de los lenguajes modernos son fuertemente tipados, es decir que obligan a indicar siempre el tipo de todas las variables antes de usarlas (declararlas). Java es fuertemente tipado.

Cabe destacar el hecho de que **muchos lenguajes disponen de una sentencia nula**, que se puede usar para colocar en cualquier lugar donde la sintaxis del programa requiera de una sentencia pero no queramos que esa sentencia tenga ningún efecto. La sentencia nula en Java es el punto y coma sin más (**;**) No obstante, si las cosas se hacen bien, siempre puede evitarse el uso de este tipo de sentencias.

5.3 Sentencias de control de flujo.

En todos los lenguajes de programación son necesarias ciertas sentencias de control del flujo de ejecución del programa, que indiquen el orden de ejecución de las sentencias, bajo qué condiciones deben ejecutarse o no, o si deben ejecutarse repetidas veces.

Ésta es la función de las **sentencias de control de flujo, que pueden considerarse como estructuras dentro del programa**. Que son básicamente 3:

- **Secuencial.**
- **Condicional o selectiva.**
- **Cíclica, repetitiva o iterativa.**

Aunque en cada lenguaje puede haber, y de hecho hay, más de una sentencia para cada tipo.

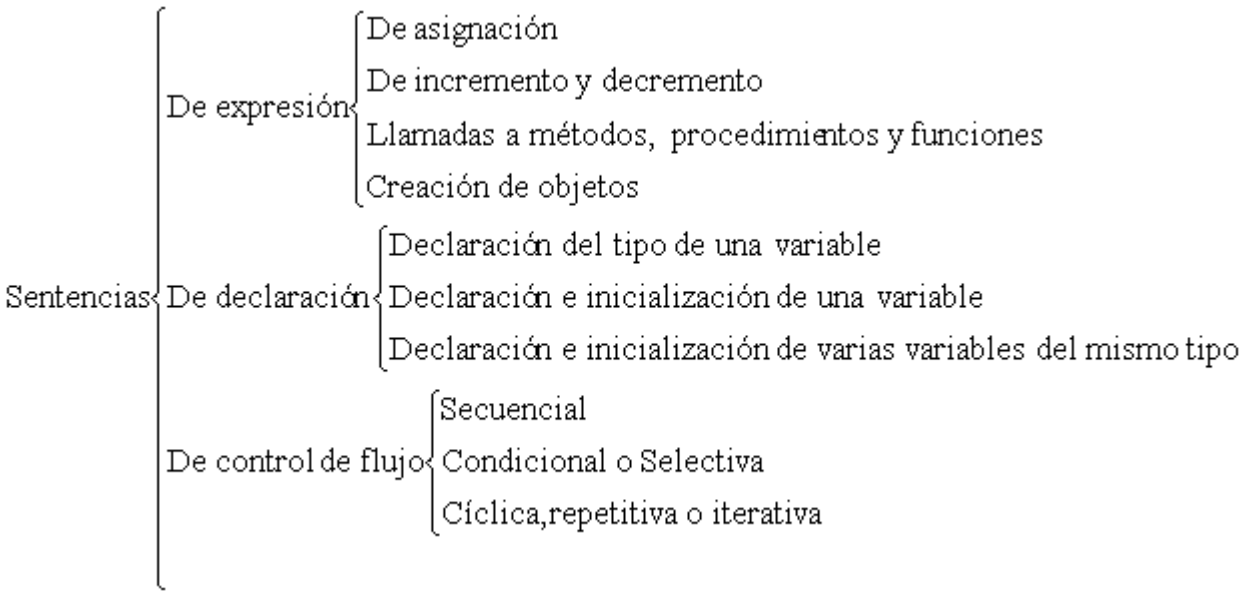
Por último añadir que en cualquier parte en que la sintaxis de un lenguaje de programación permita colocar una sentencia, también se podrá colocar todo un bloque de sentencias, delimitado de alguna forma, mediante algún símbolo o palabra reservada especial.

Por ejemplo, en Java los bloques de sentencias se delimitan con llaves.

```
{
   sentencia_1;
   sentencia_2;
 ...
   sentencia_n;
}
```

5.3.1 Cuadro resumen de los tipos de sentencias

En el siguiente esquema tenemos una referencia rápida, que ayuda a recordar los conceptos vistos sobre tipos de sentencias:



6 Estructura general de un programa

Todo el código fuente Java se escribe en documentos de texto con extensión .java

Java es compatible con la codificación Unicode, que usa 16 bits (2 bytes por carácter) e incluye la mayoría de los códigos del mundo.

En la práctica significa que los programadores que usen lenguajes distintos del inglés no tendrán problemas para escribir símbolos de su idioma. Y esto se puede extender para nombres de clase, variables, etc.

Hay código ya escrito que se puede utilizar en los programas que realicemos en Java. Se importan clases de objetos que están contenidas, a su vez, en paquetes estándares. La sintaxis de esta instrucción es:

```
import paquete.subpaquete.subsubapquete....clase
```

Por ejemplo, para utilizar la clase **Date** escribiremos:

import java.util.Date

Lo que significa, importar en el código la clase **Date** que se encuentra dentro del paquete **util** que, a su vez, está dentro del gran paquete llamado **java**.

También se puede utilizar el asterisco en esta forma:

import java.util.*

Esto significa que se va a incluir en el código todas las clases que están dentro del paquete **util** de **java.**

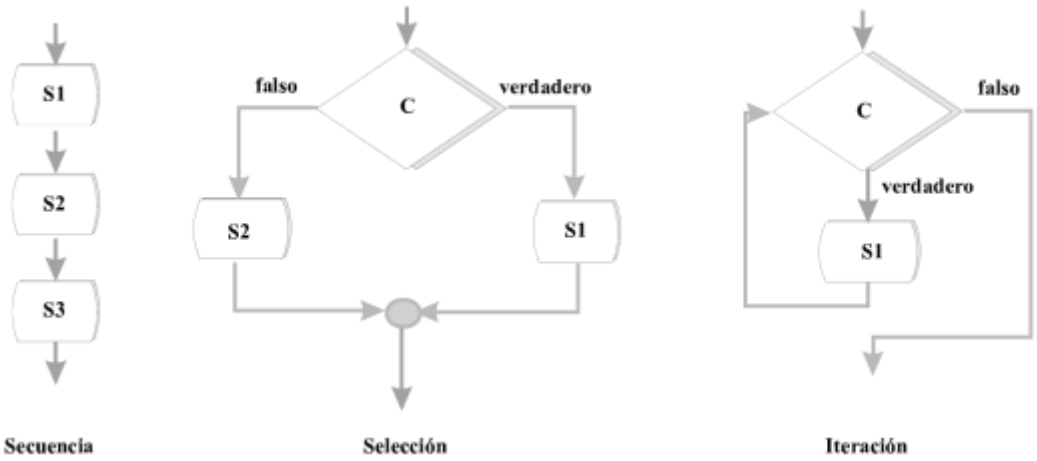
Un programa de ordenador se puede definir como una secuencia ordenada de instrucciones, dedicadas a ejecutar una tarea. Debido a esto aparece el concepto de *flujo de ejecución* de un programa, que define el orden que siguen las sentencias durante la ejecución del mismo.

El flujo de ejecución de un programa viene determinado por una serie de patrones o estructuras de programación. Cada una de estas estructuras de programación se comporta exteriormente como una sentencia única, de forma que se pueden concatenar dentro de otras estructuras y así componer el flujo de ejecución de un programa completo. Estas estructuras de programación son independientes del lenguaje de programación que se esté usando, se pueden aplicar a cualquiera de los que existen en la actualidad y son las sentencias de control de flujo:

- **Secuencial**: compuesta por 0, 1 o N sentencias que se ejecutan en el orden en que han sido escritas. Es la estructura más sencilla y sobre la que se construirán el resto de estructuras.
- **Condicional o selectiva**: consta de una sentencia especial de decisión y de un conjunto de secuencias de instrucciones. Según la evaluación de la sentencia de decisión se generará un resultado (que suele ser verdadero o falso) y en función de éste, se ejecutarán una secuencia de instrucciones u otra. Las estructuras de selección podrán ser simples, compuestas y múltiples.
- **Repetición**: consta de una sentencia especial de decisión y de una secuencia de instrucciones. La sentencia de decisión solo genera dos tipos de resultado (verdadero o falso). Si la sentencia de decisión genera un resultado correcto la secuencia de instrucciones se ejecutará de forma iterativa. Si genera un resultado incorrecto se finalizará la ejecución de la estructura de repetición.

Para evaluar las condiciones de las sentencias de decisión se utilizan los operadores relacionales y lógicos que ya conocemos.

En los ejemplos que siguen vemos el diagrama de flujo para los distintos tipos de estructuras de programación:



7 Sentencias y bloques

En Java, una **sentencia simple** debe terminar con un ';'.

Para generar una secuencia de más de una sentencia simple (también llamada **bloque**) hay que agrupar entre llaves las sentencias simples que la componen:

```
{
        sentencia_1;
        sentencia_2;
        …
        sentencia_n;
}
```

Ejemplo de programa que utiliza solo sentencias simples (Javaya.com.ar: estructura de programación secuencial):

Tenemos dos entradas num1 y num2, dos operaciones: realización de la suma y del producto de los valores ingresados y dos salidas, que son los resultados de la suma y el producto de los valores ingresados. En el símbolo de impresión podemos indicar una o más salidas, eso queda a criterio del programador, lo mismo para indicar las entradas por teclado.

```
import java.util.Scanner;

public class SumaProductoNumeros {
    public static void main(String[] ar) {
      Scanner teclado = new Scanner(System.in);
      int num1, num2, suma, producto;
      System.out.print("Introduzca primer valor:");
      num1 = teclado.nextInt();
      System.out.print("Introduzca segundo valor");
      num2 = teclado.nextInt();
      suma = num1 + num2;
      producto = num1 * num2;
      System.out.print("La suma de los dos valores es:");
```

```
        System.out.println(suma);
        System.out.print("El producto de los dos valores es:");
        System.out.println(producto);
    }
}
```

8 Estructuras de selección

8.1 Sentencia if

Utilizamos la selección (o condición) cuando deseamos que el ordenador elija entre varias acciones alternativas. Para ello, creamos una condición, una afirmación que puede ser verdadera o falsa. Si se cumple la condición se ejecutan una serie de instrucciones. Se puede hacer que otro conjunto de instrucciones se ejecute si la condición es falsa. La sintaxis de la instrucción **if** es:

```
if (condición) {
      // instrucciones que se ejecutan si la condición es true
}
else {
      // instrucciones que se ejecutan si la condición es false
}
```

La parte **else** es opcional. Ejemplo:

```
if ((diasemana>=1) && (diasemana<=5)){
      trabajar = true;
}
else {
   trabajar = false;

}
```

Si después de un if solo se ejecuta una sentencia no es necesario poner llaves. Conviene ponerlas para tener una mejor visión de los bloques que se ejecutan.

Se pueden anidar varios if a la vez. De modo que se comprueban varios valores. Ejemplo:

```
if (diasemana==1) dia = ”Lunes”;
else if (diasemana==2) dia = ”Martes”;
else if (diasemana==3) dia = ”Miércoles”;
else if (diasemana==4) dia = ”Jueves”;
else if (diasemana==5) dia = ”Viernes”;
else if (diasemana==6) dia = ”Sábado”;
else if (diasemana==7) dia = ”Domingo”;
else dia = ”?”;
```

8.2 Sentencia switch

Es la estructura condicional compleja porque permite evaluar varios valores a la vez.

Sintaxis:

```
switch (expresión) {
   case valor1:
       // sentencias si la expresión es igual al valor1;
       [break]
   case valor2:
       // sentencias si la expresión es igual al valor2;
       [break]
       .
       .
       .
   default:
      // sentencias que se ejecutan si no se cumple ninguna de las anteriores
}
```

Esta instrucción evalúa una expresión (que debe ser **short, int, byte** o **char**), y según el valor de la misma ejecuta instrucciones. Cada **case** contiene un valor de la expresión; si efectivamente la expresión equivale a ese valor, se ejecutan las instrucciones de ese **case** y de los siguientes.

1. Al contrario que otros lenguajes, no se puede utilizar como expresión de control nada que no dé como resultado un tipo entero.
2. 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**.
3. 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.
4. Es posible indicar un caso por defecto (**default**), que se activará si ninguno de los casos indicados mediante cláusulas case ha sido activado.
5. 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.

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.** De tal modo que si queremos que para un determinado valor se ejecuten las instrucciones de un apartado **case** y sólo las de ese apartado, entonces habrá que finalizar ese **case** con un **break**.

Ejemplo 1:

```
switch (diasemana) {
   case 1:
      dia = ”Lunes”;
      break;
   case 2:
      dia = ”Martes”;
      break;
```

```
        case 3:
           dia = ”Miércoles”;
           break;
        case 4:
           dia = ”Jueves”;
           break;
        case 5:
           dia = ”Viernes”;
           break;
        case 6:
           dia = ”Sábado”;
           break;
        case 7:
           dia = ”Domingo”;
           break;
        default:
           dia = ”?”;
    }
```

Ejemplo 2:

```
    switch (diasemana) {
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
           laborable = true;
           break;
        case 6:
        case 7:
           laborable = false;
    }
```

9 Estructuras de repetición

En su construcción utilizan los distintos **operadores de asignación** que ya hemos visto.

9.1 Sentencia while

Permite crear bucles. Un bucle es un conjunto de sentencias que se repiten si se cumple una determinada condición. Los bucles **while** agrupan instrucciones las cuales se ejecutan continuamente hasta que una condición que se evalúa sea falsa. A ese conjunto de instrucciones que se ejecutan lo denominamos **cuerpo del bucle**.

La condición se mira antes de entrar dentro del while y cada vez que se termina de ejecutar las instrucciones del while.

Sintaxis:

```
while (condición) {
   // sentencias que se ejecutan si la condición es true
}
```

Ejemplo (cálculo del factorial de un número, el factorial de 4 sería: 4*3*2*1):

```
// factorial de 4
int n = 4, factorial = 1, temporal = n;
while (temporal>0) {
   factorial *= temporal; // factorial = factorial * temporal
    temporal--; // temporal = temporal - 1
}
```

9.2 Fases de ejecución de un bucle

El cuerpo de un bucle se ejecuta en diferentes fases:

- El momento en que se alcanza la primera instrucción del bucle es lo que denominamos **entrada al bucle**.
- Cada vez que se ejecuta el cuerpo del bucle, se hace una pasada. A dicha pasada la denominamos **iteración**.
- Antes de cada iteración se comprueba la **condición del bucle**.
- Cuando se completa la última iteración y se ejecuta la primera instrucción que sigue al bucle decimos que se ha producido la **salida del bucle**.
- La condición que provoca la salida del bucle se denomina **condición de terminación**. En el caso de un bucle ***while***, por ejemplo, la condición de terminación es la misma que la del bucle.

En general, existen dos tipos de bucle: los **controlados por contador** (que se ejecutan un número determinado de veces) y los **controlados por condición** (que se repiten hasta que sucede algo dentro del bucle).

9.3 Sentencia do… while

Crea un bucle muy similar al anterior, en el que también las instrucciones del bucle se ejecutan hasta que una condición pasa a ser falsa. La diferencia estriba en que en este tipo de bucle la condición se evalúa después de ejecutar las instrucciones; lo cual significa que el bucle se ejecuta al menos una vez.

Sintaxis:

```
do {
   // instrucciones
} while (condición)
```

9.4 Sentencia for

Es un bucle más complejo especialmente pensado para rellenar arrays o para ejecutar instrucciones controladas por un contador. Una vez más se ejecutan una serie de instrucciones en el caso de que se cumpla una determinada condición. Sintaxis:

```
for (expresiónInicial; condición; expresiónEncadavuelta) {
   // instrucciones;
}
```

La **expresión inicial** es una instrucción que se ejecuta una sola vez: al entrar por primera vez en el bucle **for** (normalmente esa expresión lo que hace es dar valor inicial al contador del bucle).

La **condición** es cualquier expresión que devuelve un valor lógico. En el caso de que esa expresión sea verdadera se ejecutan las instrucciones. Cuando la condición pasa a ser falsa, el bucle deja de ejecutarse. La condición se valora cada vez que se terminan de ejecutar las instrucciones del bucle.

Después de ejecutarse las instrucciones interiores del bucle, se realiza la expresión que tiene lugar tras ejecutarse las instrucciones del bucle (que, generalmente, incrementa o decrementa al contador). Luego se vuelve a evaluar la condición y así sucesivamente hasta que la condición sea falsa.

Ejemplo (factorial):

```
//factorial de 4
int n=4, factorial=1, temporal;
for (temporal=n;temporal>0;temporal--){
   factorial *=temporal;
}
```

Variantes del for

- La condición no tiene por que implicar sólo al contador, puede ser más general

  Ejemplo:

  ```
  for(x = 0; x+y<10 && z!=0; x++)
  ```

- Puede haber en la tercera parte múltiples incrementos

  Ejemplo:

  ```
  for(x = 0; x<=10; x++,y++)
  ```

- Pueden inicializarse varias variables aunque no sean el contador

  Ejemplo:

  ```
  for(x = 0, y = 0; x<=10; x++)
  ```

10 Sentencias de salida de un bucle

10.1 break

La sentencia break ya ha aparecido asociada a la sentencia switch. También puede emplearse en el interior de sentencias iterativas. En todos los casos, permite salir inmediatamente del bucle en el que se encuentra y 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)	Alternativa estructurada al uso de break

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

10.2 continue

Instrucción que siempre va colocada dentro de un bucle y que hace que el flujo del programa ignore el resto de instrucciones del bucle; dicho de otra forma, va hasta la siguiente iteración del bucle. Al igual que ocurría con **break**, hay que intentar evitar su uso.

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)	Alternativa estructurada al uso de continue

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

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. (**RECOMENDADO NO UTILIZARLO**)

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.

11 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.