INAP
INSTITUTO NACIONAL DE ADMINISTRACIÓN PÚBLICA

Curso online: **PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA (TELEFORMACIÓN - ON LINE)**

Tema 2: SINTAXIS BÁSICA DE JAVA

Autores

Jorge Molinero Muñoz

Miguel Sallent Sánchez

ÍNDICE

1. INTRODUCCIÓN

Antes de aprender los entresijos de Java como lenguaje de programación orientado a objetos y sus capacidades para desarrollar aplicaciones es necesario conocer la sintaxis básica de dicho lenguaje. Para ello, en este tema se presentan los principales conceptos de la sintaxis de Java no específicamente ligados a la programación orientada a objetos, si bien en algunos puntos será necesario realizar alguna mención a elementos de la misma. Los elementos de sintaxis relacionados con la **Programación Orientada a Objetos** (POO) se presentarán en el tema siguiente.

La sintaxis de Java está totalmente inspirada en la de C y C++, por lo que para aquellas personas familiarizadas con estos lenguajes este tema resultará sencillo. Sin embargo, en Java se han omitido algunos elementos presentes en la sintaxis de C, entre ellos los problemáticos *punteros*, puesto que el uso incorrecto de estos provoca numerosos errores en el código. Aunque es cierto que los descriptores de objeto de Java están implementados internamente como punteros, el programador de Java no tiene la capacidad de manipularlos directamente, por tanto, no se puede, convertir un entero en un puntero ni referenciar una dirección de memoria arbitraria.

Asimismo, desaparecen elementos como los *struct de C*, por la sencilla razón de que en realidad las estructuras no son sino tipos de datos compuestos que se asocian directamente con objetos. A lo largo del capítulo se irán presentando nuevas características de Java.

2. VARIABLES

Las *variables* son los elementos básicos que almacenan datos en memoria. Toda variable se define mediante un nombre, un tipo y un ámbito, y va tomando distintos valores a lo largo de la ejecución del programa.

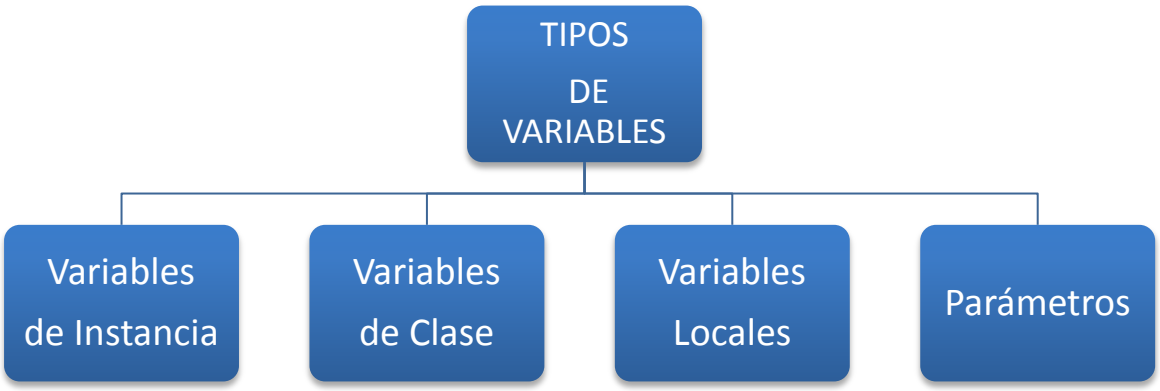
En Java se distinguen los siguientes tipos de variables:

- **Variables de Instancia** (Campos no estáticos) o atributos. Son las variables que se utilizan para almacenar la información propia de los objetos, estas pueden contener distinta información para cada una de las instancias de una clase, es decir, cada objeto de la clase pueden tener distintos valor almacenado. Estas variables se caracterizan por no ser estáticas, no les precede el modificador *static.* Por ejemplo, en el caso de representar objetos de tipo avión, la variable altura sería una variable de Instancia, porque cada avión de dicha clase puede tener una altura.

- **Variables de Clase** (Campos estáticos). Son variables que almacenan información de la clase, de modo que representa un valor que comparten todas las instancias u objetos de la clase. En el caso de la clase avión, el número de alas sería una variable de clase. Y se declaran poniendo el modificador *static.*

- **Variables Locales**. Son variables que se declaran y utilizan para almacenar valores intermedios en una sección del código, generalmente en un método o en un bloque delimitado por llaves. Una vez que el método o el bloque donde se han definido termina, la variable local deja de existir y su valor se pierde. No tienen una sintaxis especial, el nombre lo reciben por el lugar en el que se declaran.

- **Parámetros.** Son variables que se utilizan para enviar información a los métodos, permiten al desarrollador enviar datos con los cuales trabajará el método.



No existen variables globales como las entendemos en otros lenguajes de programación. Es imposible crear una variable que sea visible para todas las clases, salvo que estas clases pertenezcan a una misma jerarquía. Así, si se necesita comunicar objetos pertenecientes a una jerarquía de clases se puede recurrir a las variables de clase, aunque no se recomienda su utilización.

Declaración de una variable

Para declarar una variable se debe especificar el tipo de la variable, que puede ser un tipo básico o el nombre de una clase o interfaz (ya profundizaremos más adelante en los tipos) y su nombre:

```
tipo identificadorVariable;
```

Ejemplo: `int numeroPuertas;`

También se pueden declarar varias variables de un mismo tipo en una sola sentencia, sin más que anteponer el nombre del tipo a los nombres de variables de ese tipo separadas por comas:

```
tipo identificadorVariable1, identificadorVariable2, ...;
```

Inicialización de variables

Las variables pueden ser inicializadas en el momento de la declaración mediante la sentencia:

```
// si el tipo es un tipo básico:
tipo variable = valorInicial;

// si se trata de una variable de instancia (objeto):
NombreClase  variable = new ConstructorClase(...);
```

Se muestran a continuación ejemplos de declaraciones de variables:

Declaración simple de variables

```
int edad;
String nombre;
boolean esCierto;
```

Declaración múltiple de variables del mismo tipo

```
int x,y,z;
String nombre, apellido1, apellido2;
boolean esCierto, esFin;
```

Declaración de variables con inicialización

```
int edad= 24, i = 0;
String nombre = "Laura";
boolean esCierto = false;
```

Ya sea en la declaración (más conveniente para no olvidarnos) o más adelante en el programa, todas las variables deben inicializarse, especialmente las variables locales de los métodos, que se crean con un valor *"basura"*.

Para las variables de instancia o atributos, cuando se crea el objeto a través del constructor, Java asigna un valor inicial por omisión que depende del tipo de cada atributo: *null* para instancias de clases, *0* para variables numéricas, *'\u0000'* para caracteres y *false* para booleanos.

La declaración de variables puede ir en cualquier lugar de la implementación de un método o de una clase, siempre y cuando estén declaradas antes de referenciarse por primera vez; no obstante, para las variables locales es práctica habitual declararlas justo en el ámbito en que van a ser utilizadas. Además, no se debe declarar una variable con el mismo nombre que una de ámbito exterior.

Nombre de una variable

Se recomienda que el nombre de las variables sea representativo de su función, ya que no hay una longitud máxima para el nombre, pero siempre respetando las reglas de nombrado que se indican a continuación:

- Los nombres de variables en Java, pueden comenzar por una letra, el símbolo de guión bajo(_) o el signo de dólar ($). Después de la primera letra, pueden aparecer tanto los caracteres mencionados anteriormente como números. Sin embargo, aunque estas son la reglas que impone el compilador, hay ciertas normas que sin ser obligatorias son de uso generalizado (normas de convención):

  Normas de convención.

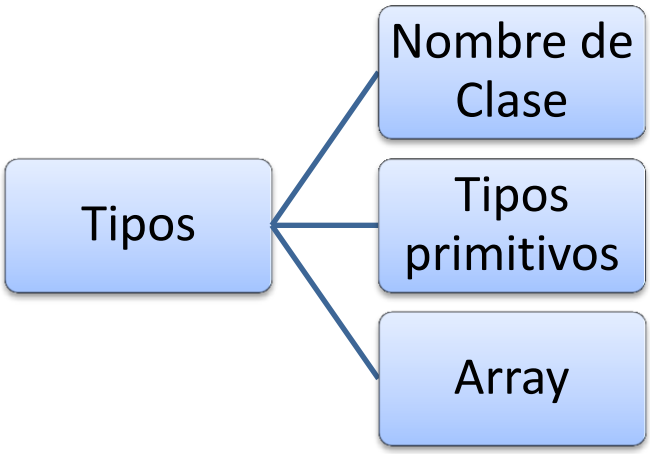
  - Las variables no deben comenzar por guión bajo (_) ni por el dólar ($)
  - Si el nombre de la variable es una única palabra, se escribiría en minúsculas. Si consiste de más de una palabra, escriba la primera letra de cada palabra en mayúsculas, excepto la primera. Los nombres capacidadActual y velocidadActual son ejemplos de esta convención.

- No se permite que los nombres de las variables contengan caracteres especiales como el espacio en blanco, %, *, @, etc, que están reservados para operadores.
- No se permite declarar una variable cuyo nombre coincida con el de alguna palabra reservada del lenguaje, como class, int, etc.
- En Java se distingue entre mayúsculas y minúsculas, por lo que las variables contador y Contador son diferentes.

Java usa el conjunto de caracteres Unicode, que además de ofrecer el conjunto de caracteres ASCII habitual, contiene otros caracteres que aparecen en diferentes idiomas, tales como caracteres acentuados, la 'ñ' española, etc. Podrán utilizarse por tanto caracteres acentuados y la 'ñ' española en la construcción de identificadores.

Tipo de una variable

El tipo de dato de una variable determina los valores que puede contener dicha variable y las operaciones que se pueden realizar sobre ella.



Java es un lenguaje muy estricto al utilizar tipos de datos. Variables de datos distintos son incompatibles. Algunos autores hablan de lenguaje fuertemente tipado o incluso lenguaje muy tipificado. Se debe a una traducción muy directa del inglés strongly typed referida a los lenguajes que, como Java, son muy rígidos en el uso de tipos.

El caso contrario sería el lenguaje C en el que jamás se comprueban de manera estricta los tipos de datos.

3. PALABRAS RESERVADAS

Las palabras reservadas son aquellos términos que tienen significado específico en el lenguaje, y por lo tanto no pueden ser utilizadas como un identificador de objetos o variables en el código. Es decir no pueden ser nombres de variables, clases o métodos.

La gramática Java específica el orden preciso en el que se puedan escribir dichas palabras reservadas y los símbolos. Y esto es utilizado por el compilador para asegurar que el programa está bien construido. Cualquier código que no esté escrito de modo correcto emitirá mensajes de error y no construirá un programa ejecutable.

A continuación se presentan un conjunto de palabras reservadas, cuyo significado se irá describiendo progresivamente a lo largo del curso:

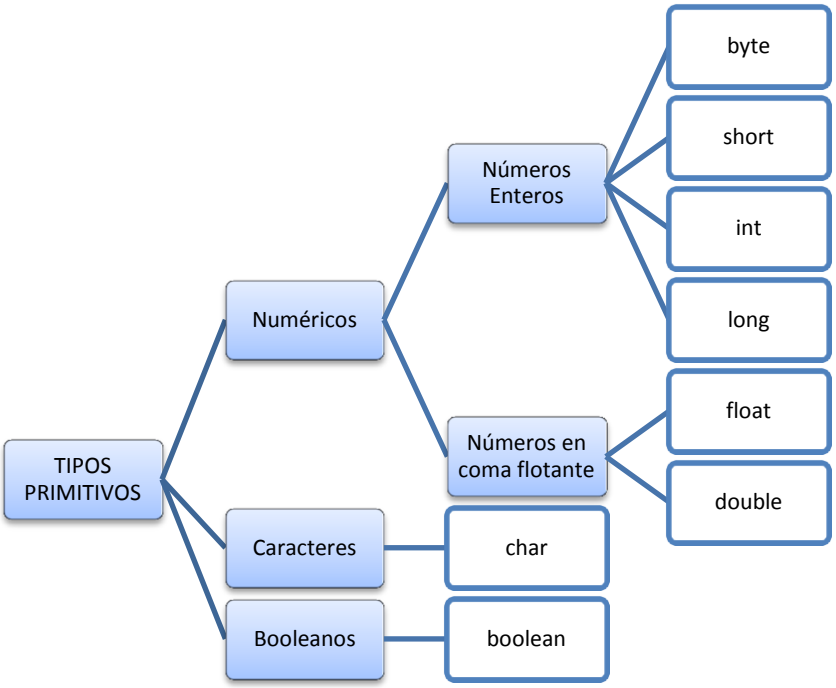
abstract	continue	for	new	switch	volatile
boolean	default	goto	null	synchronized	while
break	do	if	package	this	
byte	double	implements	private	threadsafe	
byvalue	else	import	protected	throw	
case	extends	instanceof	public	throws	
catch	false	int	return	transient	
char	final	interface	short	true	
class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

Existen palabras reservadas que no se utilizan, por ejemplo: cont y goto.

4. TIPOS PRIMITIVOS

Los tipos primitivos son un tipo de variable predefinido por el lenguaje y se nombran con una palabra reservada. Y se caracterizan porque se les puede asignar un valor directamente. Son tipos simples ya que no se tratan como objetos, para que sea más eficiente y sencillo su uso.

Existen 8 tipos primitivos en Java que serán numéricos, caracteres y booleanos



Para números enteros:

Son números sin decimales, y se les puede asignar valores enteros normales, enteros octales y hexadecimales (aunque esto es poco habitual). El tipo más conveniente se elige analizando cuál será el dominio de valores de la variable. Si se supera el valor máximo de alguno de estos tipos, se trunca.

- byte, que ocupa 1 byte y cubre el intervalo [-128 ... 127]. Es útil cuando se manejan flujos de byte, como en la lectura de un archivo o la entrada de datos de una red.
- short, 2 bytes [-32.768 ... 32.767]
- int, 4 bytes [2.147.483.648 ... 2.147.483.647]
- long, 8 bytes.

Números en coma flotante:

- float, de 32 bits.
- double, de 64 bits. Se usa para cálculos matemáticos de mucha precisión (todas las funciones matemáticas los devuelven).

Ejemplo: `double raízCuadrada;`

Como se puede observar, los tipos numéricos se caracterizan por el tamaño de dato que pueden albergar. Es posible asignar una variable de un tipo primitivo a otro siempre y cuando la variable de destino sea lo suficientemente grande para contener la variable de origen, en estos casos se realizará una conversión automática de tipos. Si no es así, se necesitará *conversión explícita (casting)*, que estudiaremos más adelante en este tema.

Caracteres:

- **El tipo carácter (char)** identifica caracteres individuales y es almacenado mediante 16 bits sin signo, debido a que emplea el conjunto de caracteres Unicode.

Ejemplo: `char primerCarácter;`

Booleanos:

- **El tipo booleano (boolean)** puede tener uno de los siguientes valores: true o false, y es el resultado que devuelven todos los operadores de comparación.

Ejemplo: `boolean terminado;`

4.1 CASTING EN TIPOS PRIMITIVOS

En ocasiones nuestros programas operan con unos tipos primitivos, y por un algún motivo, es necesario trabajar con datos de otro tipo primitivo, esto es habitual cuando se trabaja con números.

Supongamos que disponemos un programa o módulo que es capaz de calcular el mínimo común múltiplo de dos números. Este programa o módulo trabaja con números enteros de tipo int, sin embargo el programa principal opera con números enteros de tipo double. Para poder utilizar la funcionalidad del cálculo del mínimo común múltiplo, será necesario realizar un casting y convertir los números double en int, lógicamente habiendo comprobado previamente que los números recibidos no exceden la capacidad de los int.

El casting en los tipos primitivos consiste en convertir una variable de un tipo origen en otro tipo destino, con el fin de poder utilizarla en puntos del programa donde se necesite que las variables sean del tipo destino. La sintaxis es:

```
(TipoDestino) variable
```

El casting puede realizarse entre tipos básicos o en una jerarquía de herencia (que se abordará en el siguiente capítulo).

Ejemplos de casting entre tipos primitivos:

- Casting de una variable a otra variable con mayor capacidad:

```
int  cuenta1 = 0;
double  cuenta2 = (double) cuenta1; // cuenta2 tomará el valor 0.0
```

Debido a que la capacidad de la variable de destino es mayor, que la de origen, el casting no es obligatorio, se podría asignar directamente:

```
int  cuenta1 = 0;
double  cuenta2 = cuenta1; // cuenta2 tomará el valor 0.0
```

- Casting de una variable a otra variable con menor capacidad:

```
double  cuenta1 = 1.2;
int  cuenta2 = (int) cuenta1; // cuenta2 tomará el valor 1
```

En estos casos, si es obligatorio realizar el casting.

5. ARRAYS

Un array: es un grupo de variables del mismo tipo a las que nos referimos con el mismo nombre.

Para crearlo, hay que seguir 2 pasos:

1. Declarar una variable como un array indicando el tipo de los elementos del array, seguido de los corchetes vacíos [ ] y del nombre del array (también se puede poner el nombre del array seguido de los corchetes vacíos).

```
Tipo_elementos  [ ] nombreArray;
Tipo_elementos  nombreArray[ ];
```

Ejemplos:

```
char  nombre [ ]; // declara un array de caracteres.
int [ ] edades; // declara un array de enteros.
```

2. Crear propiamente el array y asignárselo a la variable declarada: para ello se usa el operador ***new*** (que estudiaremos en profundidad en el tema de Programación Orientada a Objetos) seguido del tipo de los elementos del array más la dimensión del array entre corchetes. Esto se hace así porque en realidad los arrays son objetos, aunque no estén definidos en la API.

```
nombreArray = new Tipo_elementos[dimensión];
```

Ejemplos:

```
edades = new int[10];// crea un array de 10 enteros
nombre = new char[100];// crea un array de 100 caracteres
```

Estas dos operaciones pueden hacerse simultáneamente de la siguiente forma:

```
TipoElementos  [] nombreArray = new TipoElementos[dimensión];
```

Ejemplo:

```
String [] titulos = new String[10];
```

Los valores de los elementos quedan inicializados a *0* para arrays de números, *false* para arrays de booleanos, *'\u0000'* para arrays de caracteres y *null* para arrays de cualquier otro tipo.

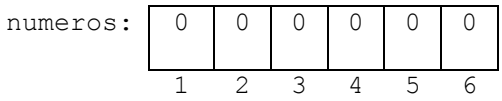
Cuando se quiere crear un array cuyos elementos ya están fijados. Es posible inicializar el contenido del array a la vez que se declara: tras declarar el array, se indica la lista de los valores iniciales de los elementos separados por comas encerrada entre llaves, sin emplear el operador *new*. La dimensión del array será igual al número de elementos que aparecen entre las llaves.

Ejemplos:

```
String[ ] titulos = { "Hamlet","Macbeth","El Quijote","Drácula"};
int[ ] edades = {18, 45, 26, 3, 22};
```

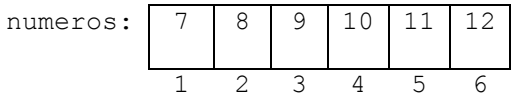
Creación de un array de 6 enteros

```
int[] numeros = new int[6];
```



Creación de un array inicializado

```
int[] numeros = {7,8,9,10,11,12};
```



Se pueden crear además arrays multidimensionales, sin más que poner entre corchetes la longitud de cada dimensión del array:

Ejemplo:

```
int matriz[ ][ ] = new int[10][20];  // Crea una matriz de 10 x 20
                                     // elementos de tipo entero
```

Para acceder a los elementos de un array, se escribe el nombre de la variable que lo representa y entre corchetes una expresión que indique la posición del elemento al que se desea acceder en el array:

nombreArray[expresiónEntera];

Ejemplo:

```
títulos[1];
títulos[i+1];     // "i" tiene que ser una variable entera
```

Se debe tener en cuenta que las posiciones de un array se empiezan a numerar desde el cero; así para un array con 10 elementos, las posiciones válidas van desde la 0 al 9. Java comprueba que el elemento al que se desea acceder está en una posición válida (no negativa y dentro del rango de valores del array), y si no lo está, genera un error en tiempo de ejecución.

6. CONSTANTES

En Java las constantes son básicamente variables cuyo valor no puede ser modificado, siempre valen lo mismo para todos los objetos de la clase, por lo tanto son variables de clase que deberán tener el modificador *static*.

Para forzar que dichas variables siempre tengan el mismo valor, y no pueda ser modificado por error, se declaran con el modificador *final*. Y debido a que no se puede cambiar el valor de la variable, es necesario inicializarla en el momento de su declaración.

A partir de estas normas, la estructura de las constantes sería:

```
static final NOMBRE_CONSTANTE = valor;
```

Por último, indicar que por convención este tipo de variables se suelen escribir en mayúsculas separando las distintas palabras por guiones bajos, por ejemplo NUM_ALAS = 2. Esto no es una norma, sólo es una convención que es conveniente respetar para hacer un código más legible.

```
static final int DIAS_SEMANA = 7;

static final int MESES_AÑO = 12;
```

7. OPERADORES

Los operadores operan con uno o más argumentos y generan un resultado. Existen diferentes tipos de operadores: *aritméticos, de comparación, lógicos, de bit y de Strings*.

- **Operadores aritméticos:** existen 5 tipos de operadores aritméticos, y sus equivalentes con asignación. El operador menos (-) puede ser usado para restar o para negar. Existen además las operaciones de pre-incremento, pre-decremento, post-incremento y post-decremento, que modifican el valor de la variable en uno antes o después de asignarle valor:

Asignaciones con operadores aritméticos		
x+=y	**Equivalente a**	**x = x + y**
x-=y	**Equivalente a**	**x = x - y**
x*=y	**Equivalente a**	**x = x * y**
x/=y	**Equivalente a**	**x = x / y**

Operadores aritméticos	
+	**Operador de adición (también se utiliza para concatenar Strings)**
-	**Operador de sustracción**
*****	**Operador de multiplicación**
/	**Operador de división**
%	**Operador de resto**

Ejemplos:

```
y=++x; //incrementa el valor de 'x' y se lo asigna a 'y'
y=x--; //se asigna 'x' a 'y' y después se decrementa el valor de 'x'
```

Aunque es poco habitual, los operadores aritméticos se pueden utilizar también con tipos char.

- **Operadores relacionales:** se usan para comparar expresiones y devuelven valores de tipo booleano (true o false).

OPERADOR	SIGNIFICADO
==	igual a
!=	distinto de
>	mayor a
>=	mayor o igual a
<	menor a
<=	menor o igual a

- **Operadores lógicos:** devuelven valores booleanos, y son:
  - AND: se representa mediante &&.
  - OR: Se representa mediante ||.
  - NOT: operador unario, se representa por ! y es la negación de la expresión.

  En los operadores && y || la segunda expresión sólo es evaluada si es necesario. Por ejemplo, con el operador || la primera expresión se evalúa siempre, y la segunda sólo si la primera es false.

- **Operadores de bit:** Realizan las operaciones descritas sobre los valores de las variables bit a bit.

OPERADOR	SIGNIFICADO
&	AND bit a bit
|	OR bit a bit
^	XOR bit a bit
~	complemento a 1
<<	desplazamiento a la izquierda con signo
>>	desplazamiento a la derecha con signo
>>>	desplazamiento a la derecha sin signo

Ejemplo:

```
a = b & c;
//'a' es igual al resultado de la operación AND entre 'b' y 'c' (bit a bit).
```

- **Operadores de Strings:** es posible concatenar dos cadenas de caracteres a través del operador *"+"*, que recibe dos operandos y crea un nuevo String resultado de la concatenación de los mismos. Si alguno de los operandos no es un *String*, se convierte, siempre que se trate de un objeto perteneciente a una clase que tenga implementado el método *toString()* (Se verá más en detalle en el tema de Orientación a Objetos). También existe el operador "*+=*", que asigna a una variable de tipo cadena el resultado de concatenarle la cadena que venga detrás de dicho operador. Así, estas dos sentencias son equivalentes:

```
nombre += "Mónica";
nombre = nombre + "Mónica";
```

Precedencia de operadores

Al igual que ocurre en el álgebra elemental, los lenguajes de programación definen una precedencia de operadores que conviene conocer, y que determina en qué orden se van a evaluar los distintos operadores que afectan a una expresión. Los operadores de un mismo nivel tienen la misma precedencia y son evaluados de izquierda a derecha según como aparezcan en la expresión.

Para variar la precedencia de estos operadores se usan paréntesis, de forma que se evalúan primero las expresiones de los paréntesis más internos. Los paréntesis contribuyen además a dotar de legibilidad al código. En el siguiente ejemplo se muestran dos expresiones, la primera sin paréntesis y la segunda con ellos. Ambas expresiones se evaluarían de la misma forma:

```
a= b*c+d/e++      a=(b*c)+(d/(e++))
```

Precedencia de operadores	
Operadores	**Precedencia**
Postfix	*expr*++ *expr*--
Unarios	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicativos	* / %
aditivos	+ -
de movimiento	<< >> >>>
relacionales	< > <= >= instanceof
de igualdad	== !=
AND a nivel de bit	&
OR exclusivo a nivel de bit	^
OR inclusivo a nivel de bit	|
AND lógico	&&

OR lógico	`||`
ternarios	`? :`
de asignación	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

8. EXPRESIONES

Una expresión es una construcción de variables, operadores e invocaciones a métodos que se puede evaluar y que devuelven un único valor. Esta construcción deberá realizarse siguiendo una sintaxis apropiada.

Un tipo particular de expresión se forma con el operador de asignación. Que básicamente devuelve el valor de la variable.

Por ejemplo:

```
x = 0;
```

El operador de asignación devuelve el valor de la variable que se encuentra a la derecha del símbolo igual *'='*, y se lo asigna a la variable *x*.

Es posible escribir sentencias más complejas con dicho operador, por ejemplo:

```
x=y=z=0;  // igual que x=0; y=0; z=0;
```

Java, permitirá construir expresiones compuestas a partir de varias expresiones más pequeñas, siempre y cuando los tipos de datos concuerden. Por ejemplo:

```
1 * 2 * 3
```

En este caso, el resultado de la expresión será un tipo primitivo numérico que resulta de multiplicar 1 por 2 y por 3.

9. COMENTARIOS

Se considera una buena práctica de programación documentar los programas incluyendo comentarios en el código fuente que expliquen el objetivo de los métodos, clases y variables del mismo.

Existen tres tipos de comentarios en Java:

- **Comentarios de una sola línea:** se pueden colocar en cualquier parte de la línea y comienza por un doble slash (//); el comentario acaba al concluir la línea.

  Por ejemplo:

```
int i=0; // Declaración de una variable
//String a; Esta sentencia no se ejecuta
```

- **Comentarios multilínea:** van encerrados entre "/*" y "*/". No se permiten comentarios anidados.

Por ejemplo:

```
/* Todo este texto es un comentario
que puede ocupar varias líneas */
float b;
/* El siguiente código no se ejecuta
int k;
*/
```

- **Comentarios utilizados para el sistema javadoc**, que sirve para generar documentación automática del código. Comienzan con "/**" y termina con "*/".

Por ejemplo:

```
/**
 * Comentarios del método
 * @param args Comentario del argumento
 */
public static void main(String[] args) { …
```

10. ESTRUCTURAS DE FLUJOS DE CONTROL

Son las encargadas de controlar el orden de ejecución del código en virtud de los cambios de valores en las variables, con el fin de romper la linealidad en la ejecución del código cuando sea necesario. En Java se mantienen la mayor parte de las estructuras de C y C++ aunque, como ya se comentó, se prescinde, por ejemplo, de la sentencia *goto*. A continuación se detallan estas estructuras acompañadas de los correspondientes ejemplos.

Sentencia if

La sentencia **if … else** es una instrucción de ramificación en la que se examina una condición booleana. Si la condición se cumple (la expresión booleana devuelve valor true) se ejecutarán las instrucciones correspondientes a la rama if, mientras que si no es así lo harán las correspondientes a la parte else. La sintaxis de esta sentencia es:

```
if (expresión_booleana) {
    sentencia_1;
    ...;
    sentencia_n;
} else {
    sentencia_1';
    ...;
    sentencia_n';
}
```

El diagrama de flujo corresponde con la figura siguiente:

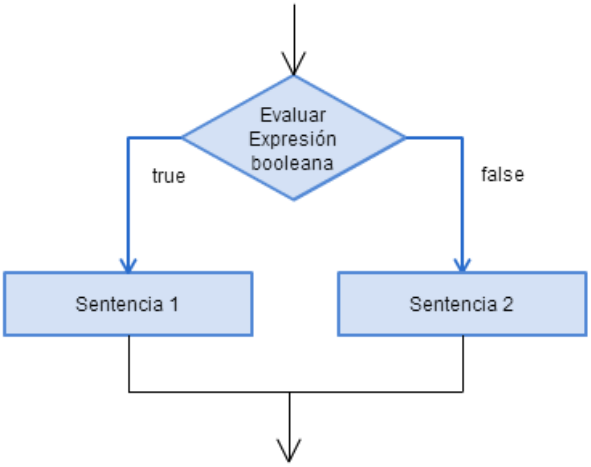
```
if (expresiónBooleana){

  Sentencia 1

} else {

  Sentencia 2

}
```



Cuando la entrada por una de las dos ramas sólo lleva asociada una sentencia el uso de llaves es opcional. La rama else no es obligatoria; si no existe y la condición if no se cumple, simplemente se continuará el flujo de ejecución con la primera sentencia que se encuentre fuera del if.

Al igual que en C, existe el *operador condicional '?'*, cuya sintaxis es:

```
(expresión booleana) ? sentencia_ExprCierta : sentencia_ExprFalsa
```

Este operador evalúa la expresión booleana situada a su izquierda; si el resultado es verdadero, se ejecuta la sentencia o grupo de sentencias situados antes de los ":", mientras que si es falso se

ejecutarán las sentencias que siguen a los ":". Se usa mucho para asignar valores a variables que dependen de una condición:

Por ejemplo:

```
int minimo = (x<y)? x : y;
```

es igual que:

```
int minimo = 0;
if (x<y)
   minimo= x;
else
   minimo = y;
```

Sentencia switch

Es otra sentencia de ramificación utilizada cuando la acción a tomar depende del valor de una determinada variable o expresión. Su sintaxis es:

```
switch (expresión) {
   case valor_1:
        operaciones_1;
        break;
   case valor_2:
        operaciones_2;
        break;
   ...
   default:
        operaciones_defecto;
}
```

La expresión puede devolver un entero o un carácter, y los valores de las ramas case deben tener el mismo tipo que la expresión. El valor de la expresión se compara con cada uno de los literales (que deben representar valores distintos). Si coincide con alguno, se ejecutarán las sentencias de su rama hasta encontrar un *break*; si no coincide con ninguno, se ejecutarán las sentencias de la rama *default*. Como esta rama es opcional, si se decide no ponerla y ningún valor coincide con el de la expresión no se hace nada.

Tampoco es obligatorio situar la sentencia *break* al final del grupo de sentencias de una rama; en este caso serán ejecutadas las sentencias correspondientes a opciones siguientes, hasta encontrar el primer *break*.

Bucle for

La sentencia **for**, junto con las sentencias **while** y **do...while**, se denominan *bucles* y su misión es repetir una sentencia o bloque de sentencias un número de veces hasta que se cumpla una determinada condición.

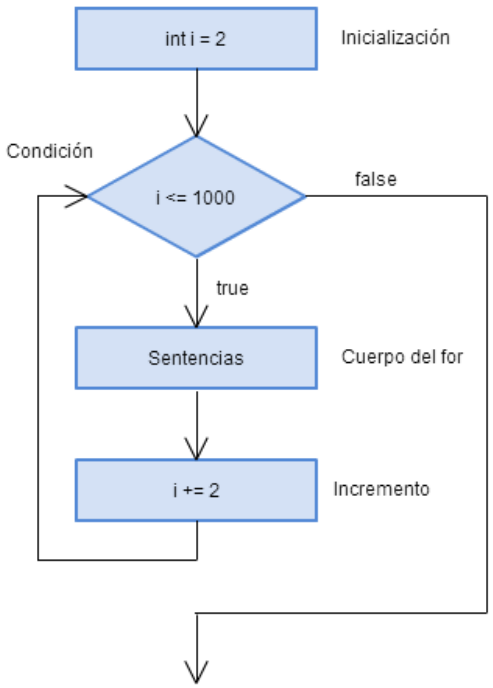
El *bucle for* tiene la siguiente sintaxis, y su diagrama de flujo es el de la siguiente figura:

Sintaxis

```
for (inicialización; condición; actualización)
{
    bloque de sentencias;
}
```

Ejemplo

```
for (int i=2;i<=1000;i+=2){
   Sentencias;
}
```



Aunque ninguna de sus partes es obligatoria, si alguna no existe, se deja su hueco pero respetando el ";" que la precede.

- **Inicialización** es una expresión que se ejecuta una única vez al comienzo del bucle. Si es necesario usar una variable índice para el mismo, se puede declarar en esta parte de la sentencia para que sea local al bucle y deje de existir posteriormente. Puede no existir cuando el bucle for depende de un índice que tiene validez desde antes en el programa.

- **Condición** es una expresión booleana, que se comprueba en cada ejecución del bucle. Si devuelve cierto, se entra de nuevo en el bucle y, si es falsa, se sale de él. Es decir, es una condición de permanencia, mientras devuelva true se seguirá ejecutando el bucle. Esta condición también se comprueba antes de la primera ejecución del bucle, de forma que si inicialmente no

se cumple la condición, el bucle no se ejecutará ninguna vez. Si no está presente, el bucle es infinito.

- **Actualización** es una expresión que se ejecuta al finalizar cada iteración, suele ser usada para modificar el valor del índice del bucle o de las variables de la condición, con el fin de poder alcanzar la condición de salida del bucle. Tampoco es obligatoria, porque la actualización de las variables puede producirse en el cuerpo del bucle.

Las partes de inicialización y actualización pueden tener varias sentencias, que deben estar separadas por comas.

Ejemplo:

```
for (int i=0; i<10; i++) {
   bloque de sentencias;
}
```

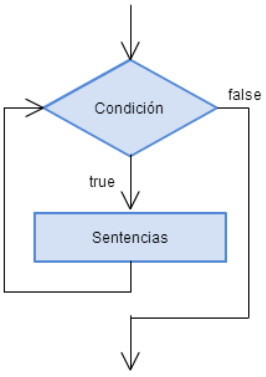
En este ejemplo el bloque de sentencias se ejecuta 10 veces: inicialmente el contador vale cero, al final de cada iteración se incrementa una unidad, y el bucle se ejecuta mientras el contador es menor que 10.

Un ejemplo claro de utilización de esta sentencia es recorrer el contenido de un array realizando algún tratamiento con sus elementos.

Bucle while

El *bucle while* se usa para repetir la ejecución de la sentencia o bloque de sentencias mientras que se cumpla la condición que le sigue. Es similar al bucle for pero no tiene las sentencias de inicialización ni actualización. La sintaxis del bucle while es la siguiente (su diagrama de flujo, a modo de ejemplo, se muestra en la siguiente figura):

```
while (condición) {
   bloque de sentencias;
}
```

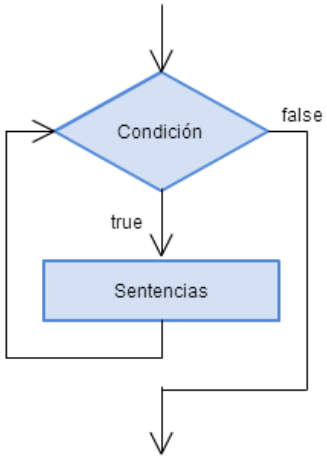


El bloque de sentencias se puede ejecutar de *0 a n veces* (tantas como veces sea cierta la condición).

Bucle do ... while

El *bucle do...while* funciona exactamente igual que el bucle while, es decir ejecuta las sentencias mientras se cumpla una condición, diferenciándose únicamente en que el bloque de sentencias es ejecutado como mínimo una vez, ya que la condición se examina después de la primera entrada en el bucle.

```
do {
    bloque de sentencias
} while (condición);
```



Sentencias break y continue

Se ha visto anteriormente que para poder abandonar la ejecución de un bucle, es necesario que la condición de salida del mismo se cumpla. Hay ocasiones en que es necesario salir de un bucle sin esperar a que se cumpla la condición (por ejemplo, si se llega a una situación de error), y para ello Java proporciona las sentencias *break* y *continue*.

- **La sentencia break** provoca la salida del bloque más interno que la contenga; así, si tenemos varios bucles anidados, la ejecución de un break provocará la salida del bucle en curso hacia el bucle inmediatamente anterior (en cuanto a anidamiento). Esta sentencia aparece asiduamente en las ramas case de una estructura *switch*, como ya se comentó.

  Ejemplo:

```
for (int i=1; (i<5); i++)  {
  for (int j=1; (j<5); j++)  {
    if (j==3)
       break; }  //salimos al primer for
}
```

- **La sentencia continue** provoca que se dejen de ejecutar las sentencias de su bloque, se regresa al punto en que se comprueba la condición y se evalúa: si se cumple, ejecuta una nueva iteración, y si no se cumple sale del bucle.

Es posible *"etiquetar" l*os bucles de tal manera que, al usar las sentencias *break* o *continue*, se pueda indicar a qué sentencia se desea saltar. Para etiquetar una sentencia se emplea la sintaxis:

etiqueta : sentencia;

Ejemplo:

```
externo:for (int i=0; i<10; i++)  {
   for (int j=0; j<10; j++)  {
     if (j>i) {
       System.out.println(" ");
     continue externo;
   }
   System.out.println(" " + (i*j) ); } //for
} //for
```

Finalmente debemos decir que las instrucciones *break* y *continue* empeoran la legibilidad del código y su mantenimiento, y deberemos evitarlas en la medida de lo posible.

Sentencia return

Como veremos más adelante, *Java* utiliza lo que se denominan métodos para acceder a los objetos. Los métodos actúan de manera similar a las funciones o procedimientos de otros lenguajes de programación, por lo que es necesario invocarlos y en ocasiones tienen que devolver un valor, retornando al punto en que fueron llamados.

Para ello se utiliza la sentencia **return**, cuya sintaxis es:

```
return  valor;    // si el método devuelve algún dato
return;          // si queremos salir sin devolver valor
```

11. EL MÉTODO MAIN

Toda aplicación debe tener una clase principal (que puede ser public o protected) en la que deberá estar definido un método denominado **main**, que es el primero que se ejecuta cuando se inicia la aplicación desde la línea de comandos o consola. Este método es uno de los pocos elementos que rompe con una metodología completamente orientada a objetos en Java. Su presencia se debe a que la aplicación debe comenzar a ejecutar por algún punto concreto del código.

Es importante destacar que la definición del método main siempre debe ser la que se indica a continuación:

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

El significado de estas palabras se estudiará en profundidad en el tema de *Programación Orientada a Objetos*, pero de momento presentamos una breve descripción:

- **public:** el método debe ser público ya que debe ser visible en todo el programa y desde el exterior, para poder ser ejecutado por la máquina virtual de Java.

- **static:** indica que se aplica a toda la clase y no a objetos o instancias de la misma. Como este método es el primero que se ejecuta, lo hace antes de que se haya creado ninguna instancia de ninguna clase, de ahí su naturaleza static.

- **void:** al ser el primer método en ejecutarse no puede devolver ningún valor, ya que no existe código que pueda recibir ese valor. En el caso de que quiera devolverse un valor en este método, debería usarse el método *exit()* de la clase *java.lang.System*.

- **args:** el parámetro del método es un *array de Strings*, que se emplea para pasarle a la aplicación parámetros mediante la línea de comandos. Los argumentos deben ir separados por blancos, y si se desea pasar una cadena de caracteres con espacios en blanco tendremos que entrecomillarla. Una consideración a tener en cuenta es que args actúa como el argv[ ][ ] de C, pero en lugar de guardar en la primera posición del array el nombre del programa (como ocurre en C) guarda el primer argumento. Asimismo, en C el número de argumentos introducidos viene dado por la variable argc, mientras que en Java se obtiene mediante un atributo del array denominado args.length.

12. EJERCICIOS PRÁCTICOS

12.1 EJERCICIO RESUELTO: PRODUCTO ESCALAR

Datos dos vectores (x1,y1,z1), (x2,y2,z2), su producto escalar se define como x1*x2 + y1*y2 + z1*z2.

Se desea realizar un programa que calcule y muestre el producto escalar de dos vectores de dimensión conocida, con la particularidad de que si alguna componente de ellos tiene valor 0 hay que indicarlo lanzando un mensaje.

Para probar su funcionamiento crearemos 3 vectores, dos de ellos con componentes no nulas y uno en que la tercera componente sea 0.

Con el fin de que el lector se familiarice con la sintaxis de Java, se recomienda multiplicar cada par de vectores a través de un bucle diferente.

```
public class EjercicioVectores {

    public static void main(String args[]) {

        int[] vector1 = {1,2,3};
        int[] vector2 = {4,5,6};
        int[] vector3 = {3,2,0};

        boolean hayCero = false;
        int resultado = 0, i = 0;

        for (i=0; i<3 && !hayCero; i++){
            hayCero = vector1[i] == 0 || vector2[i] == 0;
            resultado += vector1[i] * vector2[i];
        }

        if (!hayCero){
            System.out.println("El producto escalar de v1 por v2 vale: " + resultado);
        } else {
            System.out.println("Alguna componente de v1 o v2 era 0");
        }

        i=resultado=0;
        hayCero = false;

        while (i<3 && !hayCero){
            hayCero = vector1[i] == 0 || vector3[i] == 0;
            resultado += vector1[i] * vector3[i];
            i++;
        } //while

        if (!hayCero){
            System.out.println("El producto escalar de v1 por v3 vale: " + resultado);
        } else {
            System.out.println("Alguna componente de v1 o v3 era 0");
        }

    }//main

}//class
```