CLASE 3. PROGRAMACION EN JAVA

En esta clase analizaremos la sintaxis del lenguaje Java y comenzaremos a hacer nuestros primeros programas.

3.1 VARIABLES

Definición:

Una variable es un nombre que contiene un valor que puede cambiar a lo largo del programa.

De acuerdo con el tipo de información que contienen, en Java hay dos tipos principales de variables:

1) Variables de tipos primitivos. Están definidas mediante un valor único.

2) Variables referencia. Las variables referencia son referencias o nombres de una información más compleja: arrays u objetos de una determinada clase.

En este capítulo únicamente analizaremos las variables de tipos primitivos, mientras que las **variables referencia** serán presentada en la próxima clase para que puedan estudiarlas.

Desde el punto de vista de su papel en el programa, las variables las podemos ver como:

1) Variables miembro de una clase: Se definen en una clase, fuera de cualquier método; pueden ser tipos primitivos o referencias.

2) Variables locales: Se definen dentro de un método o, en general, dentro de cualquier bloque entre llaves {}. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también tipos primitivos o referencias.

En Java, una variable está formada por un nombre o identificador y un tipo, una declaración sería :

```
tipoVariable nombre;
int a;
float b;
boolean c;
String s;
```

Todas las variables deben tener un tipo de dato. El tipo de la variable determina los valores que la misma puede contener y las operaciones que se pueden realizar con ella.

3.1.1 Identificadores

Los identificadores nombran variables, funciones, clases y objetos; cualquier cosa que el programador necesite identificar o usar.

En Java, un identificador comienza con una letra, un subrayado (_) o un símbolo de peso ($) y continua con cualquier otro símbolo que no sea un espacio en blanco. Los

siguientes caracteres pueden ser letras o dígitos. Se distinguen las mayúsculas de las minúsculas y no hay longitud máxima.
Además, deben tomar en cuenta las siguientes reglas para elegir un identificador:

1) debe ser un identificador legal de Java comprendido en una serie de caracteres Unicode. Unicode es un sistema de codificación que soporta texto escrito en distintos lenguajes humanos, permite la codificación de 34.168 caracteres. Esto le permite utilizar en sus programas Java varios alfabetos como el Japonés, el Griego, el Ruso o el Hebreo. Esto es importante para que los programadores pueden escribir código en su lenguaje nativo.

2) no puede ser el mismo que una palabra clave (ver 3.1.2) o el nombre de un valor booleano (true or false).
3) no deben tener el mismo nombre que otras variables cuyas declaraciones aparezcan en el mismo ámbito (más adelante en veremos claramente que significa el ámbito o scope de una variable).

La regla número 3 implica que podría existir el mismo nombre en otra variable que aparezca en un ámbito diferente.
Por convención, en Java, los nombres de variables empiezan por un letra minúscula. Si una variable está compuesta de más de una palabra, como 'nombreDato' las palabras se ponen juntas y cada palabra después de la primera empieza con una letra mayúscula.
Serían identificadores válidos:

identificador

nombre_usuario

Nombre_Usuario

_variable_del_sistema

$transaccion

y su uso sería, por ejemplo:

```
int contador_principal;

char _lista_de_ficheros;

float $cantidad_en_Ptas;
```

3.1.2 Palabras Clave

Las siguientes son las palabras clave que están definidas en Java y que no se pueden utilizar como identificadores:

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

3.1.3 Palabras Reservadas

Además, el lenguaje se reserva unas cuantas palabras más, las cuáles no se puede hacer uso como Identificadores, pero que hasta ahora no tienen un cometido específico. Son:

cast	future	generic	inner
operator	outer	rest	var

3.1.4 Tipos de datos Primitivos

Llamaremos tipos primitivos de variables a aquellas variables que contienen los tipos de información más habituales: valores boolean, caracteres y valores numéricos enteros o de punto flotante.
Java dispone de ocho tipos primitivos de variables: un tipo para almacenar valores true y false (boolean); un tipo para almacenar caracteres (char), y 6 tipos para guardar valores numéricos, cuatro tipos para enteros (byte, short, int y long) y dos para valores reales de punto flotante (float y double). Los rangos y la memoria que ocupa cada uno de estos tipos se muestran en la siguiente.

Tipo de variable	Descripción
Bolean	1 byte. Valores true y false
Char	2 bytes. Unicode. Comprende el código ASCII
Byte	1 byte. Valor entero entre -128 y 127
Short	2 bytes. Valor entero entre -32768 y 32767
Int	4 bytes. Valor entero entre -3.147.483.648 y 3.147.483.647
Long	8 bytes. Valor entre -9.223.373.036.854.775.808 y 9.223.373.036.854.775.807
Flota	4 bytes (entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de

	1.401298E-45 a 3.402823E38
Double	8 bytes (unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Tabla 3.1: Tipos primitivos de variables en Java.

Los tipos primitivos de Java tienen algunas características importantes resumiremos a continuación:

1) El tipo boolean no es un valor numérico: sólo admite los valores true o false. El tipo boolean no se identifica con el igual o distinto de cero. El resultado de la expresión lógica que aparece como condición en un bucle o en una bifurcación debe ser boolean.

2) El tipo char contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter.

3) Los tipos byte, short, int y long son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos.

4) Los tipos float y double son valores de punto flotante (números reales) con 6-7 y 15 cifras decimales equivalentes, respectivamente.

5) Se utiliza la palabra void para indicar la ausencia de un tipo de variable determinado.

3.1.5 ¿Cómo se definen e inicializan las variables?

Una variable se define especificando el tipo y el nombre de la variable. Estas variables pueden ser tanto de tipos primitivos como referencias a objetos de alguna clase perteneciente al API de Java o generada por el usuario. Las variables primitivas se inicializan a cero (salvo boolean y char, que se inicializan a false y '\0') si no se especifica un valor en su declaración. Análogamente las variables de tipo referencia son inicializadas por defecto a un valor especial: null.

Ejemplos de declaración e inicialización de variables de tipos primitivos:

```
int x;          // Declaración de la variable primitiva x. Se inicializa a 0
int y = 5;      // Declaración de la variable primitiva y. Se inicializa a 5
```

3.1.6 Ámbito y vida de las variables

Entenderemos por visibilidad, ámbito o scope de una variable, la parte de la aplicación donde dicha variable es accesible y por lo tanto puede ser utilizada en cualquier expresión.

En Java todas las variables deben estar incluidas en una clase. En general las variables declaradas dentro de unas llaves {}, es decir dentro de un bloque, son visibles y existen dentro de estas llaves.
Por ejemplo las variables declaradas al principio de una función existen mientras se ejecute la función; las variables declaradas dentro de un bloque no serán válidas al finalizar las sentencias correspondientes a dicho bloque y las variables de una clase son válidas mientras existe el objeto de la clase.

3.2 OPERADORES DE JAVA

Java es un lenguaje rico en operadores. Veremos brevemente estos operadores en los apartados siguientes.

3.3.1 Operadores aritméticos

Son operadores binarios (requieren siempre dos operandos) los que realizan las operaciones aritméticas habituales:
suma (+)
resta (-)
multiplicación (*)
división (/)
resto de la división (%).

3.3.2 Operadores de asignación

Los operadores de asignación permiten asignar un valor a una variable. El operador de asignación por excelencia es el operador igual (=). La forma general de las sentencias de asignación con este operador es:

variable = expression;

Java dispone de otros operadores de asignación. Se trata de versiones abreviadas del **operador (=) que realizan operaciones "acumulativas" sobre una variable.**

La **Tabla 3.2** muestra estos operadores y su equivalencia con el uso del operador igual (=).

Operador	Utilización	Expresión equivalente
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2

Tabla 3.3. Otros operadores de asignación.

3.3.3 Operadores unarios

Los operadores más (+) y menos (-) unarios sirven para mantener o cambiar el signo de una variable, constante o expresión numérica. Su uso en Java es el estándar de estos operadores.

3.3.4 Operador condicional (?:)

Este operador, permite realizar bifurcaciones condicionales sencillas. Su forma general es la siguiente:

Expresiónbooleana ? res1 : res2

donde se evalúa Expresiónbooleana y se devuelve res1 si el resultado es de la expresión es true y res2 si el resultado es false. Es el único operador ternario (tres argumentos) de Java. Como todo operador que devuelve un valor puede ser utilizado en una expresión. Por ejemplo las sentencias:

```
x=1;
y=10;
z = (x<y) ? x+3 : y+8;
```

asignarían a z el valor 4, es decir x+3.

3.3.5 Operadores incrementales

Java dispone del operador incremento (++) y decremento (--). El operador (++) incrementa en una unidad la variable a la que se aplica, mientras que (--) la reduce en una unidad. Estos operadores se pueden utilizar de dos formas:

> **1)** Precediendo a la variable (por ejemplo: ++i). En este caso primero se incrementa la variable y luego se utiliza (ya incrementada) en la expresión en la que aparece.
>
> **2)** Siguiendo a la variable (por ejemplo: i++). En este caso primero se utiliza la variable en la expresión (con el valor anterior) y luego se incrementa.

En muchas ocasiones ustedes podrán reconocer que estos operadores se utilizan para incrementar una variable fuera de una expresión. En este caso ambos operadores son equivalente. Si se utilizan en una expresión más complicada, el resultado de utilizar estos operadores en una u otra de sus formas será diferente. La actualización de contadores en bucles es una de las aplicaciones más frecuentes de estos operadores

3.3.6 Operadores relacionales

Los operadores relacionales sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor boolean (true o false) según se cumpla o no la relación considerada. La Tabla 3.3 muestra los operadores relacionales de Java.

Operador	Utilización	El resultado es true
>	op1 > op2	si op1 es mayor que op2

>=	op1 >= op2	si op1 es mayor o igual que op2
<	op1 < op2	si op1 es menor que op2
<=	op1 <= op2	si op1 es menor o igual que op2
==	op1 == op2	si op1 y op2 son iguales
!=	op1 != op2	si op1 y op2 son diferentes

Tabla 3.3. Operadores relacionales.

Estos operadores se utilizan con mucha frecuencia en las bifurcaciones y en los bucles, que veremos en próximos apartados de este capítulo.

3.3.7 Operadores lógicos

Los operadores lógicos se utilizan para construir expresiones lógicas, combinando valores lógicos (true y/o false) o los resultados de los operadores relacionales. La Tabla 3.4 muestra los operadores lógicos de Java. Debe notarse que en ciertos casos el segundo operando no se evalúa porque no es necesario (si ambos tienen que ser true y el primero es false ya se sabe que la condición de que ambos sean true no se va a cumplir). Esto puede traer resultados no deseados y por eso se han añadido los operadores (&) y (|) que garantizan que los dos operandos se evalúan siempre.

Operador	Nombre	Utilización	Resultado
&&	AND	op1 && op2	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
||	OR	op1 || op2	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
!	negación	! op	true si op es false y false si op es true
&	AND	op1 & op2	true si op1 y op2 son true. Siempre se evalúa op2
|	OR	op1 | op2	true si op1 u op2 son true. Siempre se evalúa op2

Tabla 3.4. Operadores lógicos.

3.3.8 Operador de concatenación de cadenas de caracteres (+)

El operador más (+) se utiliza también para concatenar cadenas de caracteres. Por ejemplo, para escribir una cantidad con un rótulo y unas unidades puede utilizarse la sentencia:

```
System.out.println("El total asciende a " + result + " unidades");
```

donde el operador de concatenación se utiliza dos veces para construir la cadena de caracteres que se desea imprimir por medio del método println(). La variable numérica result es convertida automáticamente por Java en cadena de caracteres para poderla concatenar. En otras ocasiones se deberá llamar explícitamente a un método para que realice esta conversión.

3.3.9 Precedencia de operadores

El orden en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. Por ejemplo, el resultado de x/y*z depende de qué operación (la división o el producto) se realice primero. La siguiente lista muestra el orden en que se ejecutan los distintos operadores en una sentencia, de mayor a menor precedencia:

Operadores posfijos [ ] . (params) expr++ expr--
Operadores Unarios ++expr --expr +expr -expr ~ !
Creación o cast new (type) expr
Multiplicativos * / %
Aditivos + -
Flujo << >> >>>
Relacionales < > <= >= instanceof
Igualdad == ! =
Operador AND &
OR exclusivo ^
OR inclusivo |
AND logico &&
OR logico ||
Condicional ? :
Asignación = += -= *= /= %= &= ^= |= <<= >>= >>>=

En Java, todos los operadores binarios, excepto los operadores de asignación, se evalúan de izquierda a derecha. Los operadores de asignación se evalúan de derecha a izquierda, lo que significa que el valor de la izquierda se copia sobre la variable de la derecha.

3.3.10 Separadores

En el código Java sólo hay un par de secuencias con otros caracteres que pueden aparecer; son los separadores simples, que van a definir la forma y función del código.
Los separadores admitidos en Java son:

> **()** - paréntesis. Para contener listas de parámetros en la definición y llamadas a métodos. También se utiliza para definir precedencia en expresiones, contener expresiones para control de flujo y rodear las conversiones de tipo.
>
> **{}** - llaves. Para contener los valores de matrices inicializadas automáticamente. También se utiliza para definir un bloque de código, para clases, métodos y ámbitos locales.

[] - corchetes. Para declarar tipos matriz. También se utiliza cuando se referencian valores de matriz.

; - punto y coma. Separa sentencias.

, - coma. Separa identificadores consecutivos en una declaración de variables. También se utiliza para encadenar sentencias dentro de una sentencia for.

. - punto. Para separar nombres de paquete de subpaquetes y clases. También se utiliza para separar una variable o método de una variable de referencia.

3.3 ESTRUCTURAS DE PROGRAMACIÓN

Para la explicación de este apartado consideramos que ustedes tienen algunos conocimientos de programación y por lo tanto no les explicaremos los conceptos que aparecen en profundidad.
Las estructuras de programación o estructuras de control permiten tomar decisiones y realizar un proceso repetidas veces. Son los denominados bifurcaciones y bucles. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a concepto, aunque su sintaxis varía de un lenguaje a otro.

3.3.1 Sentencias o expresiones

Una expresión es un conjunto de variables unidos por operadores. Son órdenes que se le dan al computador para que realice una tarea determinada.
Una sentencia es una expresión que acaba en punto y coma (;). Se permite incluir varias sentencias en una línea, aunque lo habitual es utilizar una línea para cada sentencia. Por ejemplo:
i = 0; j = 5; x = i + j; // Línea compuesta de tres sentencias

3.3.2 Comentarios

Existen dos formas diferentes de introducir comentarios entre el código de Java, en realidad son tres, como pronto verán.
Los comentarios son tremendamente útiles para poder entender el código utilizado, facilitando de ese modo futuras revisiones y correcciones. Además permiten que cualquier persona distinta al programador original pueda comprender el código escrito de una forma más rápida.
Les recomendamos que se acostumbren a comentar el código desarrollado, de esta forma se simplificará también la tarea de estudio y revisión posteriores.
Java interpreta que todo lo que aparece a la derecha de dos barras "//" en una línea cualquiera del código es un comentario del programador y no lo tiene en cuenta. El comentario puede empezar al comienzo de la línea o a continuación de una instrucción que debe ser ejecutada. La segunda forma de incluir comentarios consiste en escribir el texto entre los símbolos /*…*/. Este segundo método es válido para comentar más de una línea de código. Por ejemplo:

```
// Esta línea es un comentario
int a=1; // Comentario a la derecha de una sentencia
// Esta es la forma de comentar más de una línea utilizando
```

// las dos barras. Requiere incluir dos barras al comienzo de cada línea
/* Esta segunda forma es mucho más cómoda para comentar muchas líneas ya que sólo requiere marcar el comienzo y el final. */

En Java existe además una forma especial de introducir los comentarios (utilizando /**…*/ más algunos caracteres especiales) que permite generar automáticamente la documentación sobre las clases y paquetes desarrollados por el programador. Una vez introducidos los comentarios, el programa javadoc.exe (incluido en el JDK) genera de forma automática la información de forma similar a la presentada en la propia documentación del JDK. La sintaxis de estos comentarios y la forma de utilizar el programa javadoc.exe se verá en los últimos capítulos de nuestro curso.

3.3.3 Bifurcaciones

Las bifurcaciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el flujo de ejecución de un programa. Existen dos bifurcaciones diferentes: if y switch.

3.3.3.1 Bifurcación if

Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor true).

Tiene la forma siguiente:

```
if (Expresiónbooleana) {
sentencias;
}
```

Las llaves {} sirven para agrupar en un bloque las sentencias que se han de ejecutar, y no son necesarias si sólo hay una sentencia dentro del if.

3.3.3.2 Bifurcación if else

Análoga a la anterior, de la cual es una ampliación. Las sentencias incluidas en el else se ejecutan en el caso de no cumplirse la expresión de comparación (false), su sintaxis es la siguiente:

```
if (Expresiónbooleana) {
sentencias1;
}
else {
sentencias2;
}
```

3.3.3.3 Bifurcación if elseif else

Permite introducir más de una expresión de comparación. Si la primera condición no se cumple, se compara la segunda y así sucesivamente. En el caso de que no se cumpla ninguna de las comparaciones se ejecutan las sentencias correspondientes al else.

```
if (Expresiónbooleana1) {
sentencias1;
}
else if (Expresiónbooleana2) {
sentencias2;
}
else if (Expresiónbooleana3) {
sentencias3;
}
else {
sentencias4;
}
```

3.3.3.4 Sentencia switch

Se trata de una alternativa a la bifurcación if elseif else cuando se compara la misma expresión con distintos valores. Su forma general es la siguiente:

```
switch (expresión) {
case valor1: sentencias1; break;
case valor2: sentencias2; break;
case valor3: sentencias3; break;
case valor4: sentencias4; break;
case valor5: sentencias5; break;
case valor6: sentencias6; break;
[default: sentencias7;]
}
```

Las características más relevantes de switch son las siguientes:

1) Cada sentencia case se corresponde con un único valor de expresión. No se pueden establecer rangos o condiciones sino que se debe comparar con valores concretos, por ejemplo número reales. El tipo de expresión debe ser char, int o boolean.

2) Los valores no comprendidos en ninguna sentencia case se pueden gestionar en default, que es opcional.

3) En ausencia de break, cuando se ejecuta una sentencia case se ejecutan también todas las que van a continuación, hasta que se llega a un break o hasta que se termina el switch.

Ejemplo:

```
switch (c) {

        case 'a': // Se compara con la letra a
        case 'e': // Se compara con la letra e
        case 'i': // Se compara con la letra i
        case 'o': // Se compara con la letra o
        case 'u': // Se compara con la letra u
        System.out.println(" Es una vocal "); break;
        default: System.out.println(" Es una consonante ");

}
```

En este ejemplo vemos que al no poner break luego dentro de cada uno de los case, continúa la ejecución de cada uno de ellos hasta llegar a la sentencia System.out.println(" Es una vocal "); que imprime un cartel en la pantalla y luego sí está el break.

3.3.4 Bucles

Un bucle se utiliza para realizar un proceso repetidas veces. Se denomina también lazo o loop. El código incluido entre las llaves {} (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalizar el bucle (Expresiónbooleana) no se llega a cumplir nunca. Se trata de un fallo muy típico.

3.3.4.1 Bucle while

Las sentencias se ejecutan mientras Expresiónbooleana sea true.

```
while (Expresiónbooleana) {
sentencias;
}
```

3.3.4.2 Bucle for

La forma general del bucle for es la siguiente:

```
for (inicialización; Expresiónbooleana; incremento) {
sentencias;
}
```

que es equivalente a utilizar while en la siguiente forma,

```
inicialización;
while (Expresiónbooleana) {
sentencias;
```

```
incremento;
}
```

La sentencia o sentencias inicialización se ejecuta al comienzo del for, e incremento después de sentencias. La Expresiónbooleana se evalúa al comienzo de cada iteración; el bucle termina cuando la expresión de comparación toma el valor false. Cualquiera de las tres partes puede estar vacía. La inicialización y el incremento pueden tener varias expresiones separadas por comas.

Por ejemplo, el código situado a la izquierda produce la salida que aparece a la derecha:

Código:	**Salida:**

for(int i = 1, j = i + 10; i < 5; i++, j = 2*i) {
System.out.println(" i = " + i + " j = " + j);
}
	i = 1 j = 11
	i = 2 j = 4
	i = 3 j = 6
	i = 4 j = 8

3.3.4.3 Bucle do while

Es similar al bucle while pero con la particularidad de que el control está al final del bucle (lo que hace que el bucle se ejecute al menos una vez, independientemente de que la condición se cumpla o no). Una vez ejecutados los sentencias, se evalúa la condición: si resulta true se vuelven a ejecutar las sentencias incluidas en el bucle, mientras que si la condición se evalúa a false finaliza el bucle.

```
do {
sentencias
} while (Expresiónbooleana);
```

3.3.4.4 Sentencias break y continue

La sentencia break es válida tanto para las bifurcaciones como para los bucles. Hace que se salga inmediatamente del bucle o bloque que se está ejecutando sin finalizar el resto de las sentencias.

La sentencia continúe se utiliza en los bucles (no en bifurcaciones). Finaliza la iteración que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del bucle).

Vuelve al comienzo del bucle y comienza la iteración siguiente a la que se produjo el continúe.

3.3.4.5 Sentencias break y continue con etiquetas

Las etiquetas permiten indicar un lugar donde continuar la ejecución de un programa después de un break o continue. El único lugar donde se pueden incluir etiquetas es justo delante de un bloque de código entre llaves {} (if, switch, do...while, while, for) y sólo se deben utilizar cuando se tiene uno o más bucles (o bloques) dentro de otro bucle y se desea salir (break) o continuar con la siguiente iteración (continue) de un bucle que no es el actual.

La sentencia break labelName por lo tanto finaliza el bloque que se encuentre a continuación de labelName. Por ejemplo, en las sentencias,

```
bucleI: // etiqueta o label
for( int i = 0, j = 0; i < 100 ; i++){
while ( true ) {
if( (++j) > 5) { break bucleI; }                    // Finaliza ambos bucles
else { break; }                                     // Finaliza el bucle interior (
while)
}
}
```

la expresión break bucleI; finaliza los dos bucles simultáneamente, mientras que la expresión break; sale del bucle while interior y seguiría con el bucle for en i. Con los valores presentados ambos bucles finalizarán con i = 5 y j = 6 (los invito a comprobarlo).

La sentencia continue (siempre dentro de al menos un bucle) permite transferir el control a un bucle con nombre o etiqueta. Por ejemplo, la sentencia,

continue bucle1;

transfiere el control al bucle for que comienza después de la etiqueta bucle1: para que realice una nueva iteración:

```
bucle1:
for (int i=0; i<n; i++) {
bucle2:
for (int j=0; j<m; j++) {
...
if (expression) continue bucle1; then continue bucle2;
...
}
}
```

3.3.4.6 Sentencia return

Otra forma de salir de un bucle (y de un método) es utilizar la sentencia return. A diferencia de continue o break, la sentencia return sale también del método o función. En el caso de que la función devuelva alguna variable, este valor se deberá poner a continuación del return (return valor;).