

CAPÍTULO 3

© F.J.Ceballos/RA-MA

ELEMENTOS DEL LENGUAJE

En este capítulo veremos los elementos que aporta Java (caracteres, secuencias de escape, tipos de datos, operadores, etc.) para escribir un programa. El introducir este capítulo ahora es porque dichos elementos los tenemos que utilizar desde el principio; algunos ya han aparecido en los ejemplos del capítulo 1 y 2. Considere este capítulo como soporte para el resto de los capítulos; esto es, lo que se va a exponer en él, lo irá utilizando en menor o mayor medida en los capítulos sucesivos. Por lo tanto, límitese ahora simplemente a realizar un estudio con el fin de informarse de los elementos con los que contamos.

PRESENTACIÓN DE LA SINTAXIS DE JAVA

Las palabras clave aparecerán en **negrita** y cuando se utilicen deben escribirse exactamente como aparecen. En cambio, el texto que aparece en *cursiva*, significa que ahí debe ponerse la información indicada por ese texto. Por ejemplo:

```
if (expresión booleana)  
    sentencia(s) a ejecutar si la expresión booleana es verdad;  
else  
    sentencia(s) a ejecutar si la expresión booleana es falsa;
```

Los corchetes “[]” indican que la información encerrada entre ellos es opcional, y los puntos suspensivos “...” que pueden aparecer más elementos de la misma forma. Por ejemplo, la sintaxis para definir una constante es:

```
final static tipo idcte1 = cte1[, idcte2 = cte2]...;
```

Cuando dos o más opciones aparecen entre llaves “{ }” separadas por “|”, se elige una, la necesaria para la expresión que se desea construir. Por ejemplo:

```
constante_entera[{1}|L].
```

CARACTERES DE JAVA

Los caracteres de Java pueden agruparse en letras, dígitos, espacios en blanco, caracteres especiales, signos de puntuación y secuencias de escape.

Letras, dígitos y otros

Estos caracteres son utilizados para formar las *constantes*, los *identificadores* y las *palabras clave* de Java. Son los siguientes:

- Letras mayúsculas de los alfabetos internacionales:
A - Z (son válidas las letras acentuadas y la Ñ)
- Letras minúsculas de los alfabetos internacionales:
a - z (son válidas las letras acentuadas y la ñ)
- Dígitos de los alfabetos internacionales, entre los que se encuentran:
0 1 2 3 4 5 6 7 8 9
- Caracteres: “_”, “\$” y cualquier carácter Unicode por encima de 00C0.

El compilador Java trata las letras mayúsculas y minúsculas como caracteres diferentes. Por ejemplo los identificadores *Año* y *año* son diferentes.

Espacios en blanco

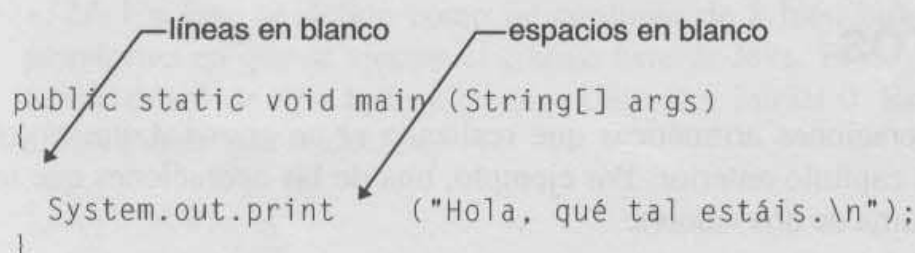
Los caracteres espacio en blanco (ASCII SP), tabulador horizontal (ASCII HT), avance de página (ASCII FF), nueva línea (ASCII LF), retorno de carro (ASCII CR) o CR LF (estos dos caracteres son considerados como uno solo: \n), son caracteres denominados *espacios en blanco*, porque la labor que desempeñan es la misma que la del espacio en blanco: actuar como separadores entre los elementos de un programa, lo cual permite escribir programas más legibles. Por ejemplo, el siguiente código:

```
public static void main (String[] args) { System.out.print(
"Hola, qué tal estáis.\n"); }
```

puede escribirse de una forma más legible así:

```
public static void main (String[] args)
{
    System.out.print("Hola, qué tal estáis.\n");
}
```

Los espacios en blanco en exceso son ignorados por el compilador. Por ejemplo, el código siguiente se comporta exactamente igual que el anterior:



```
public static void main (String[] args)
{
    System.out.print ("Hola, qué tal estáis.\n");
}
```

Caracteres especiales y signos de puntuación

Este grupo de caracteres se utiliza de diferentes formas; por ejemplo, para indicar que un identificador es una función o un array; para especificar una determinada operación aritmética, lógica o de relación, etc. Son los siguientes:

, . ; : ? ' " () [] { } < ! | / \ ~ + % & ^ * - = >

Secuencias de escape

Cualquier carácter de los anteriores puede también ser representado por una *secuencia de escape*. Una secuencia de escape está formada por el carácter \ seguido de una *letra* o de una *combinación de dígitos*. Son utilizadas para acciones como nueva línea, tabular y para hacer referencia a caracteres no imprimibles.

El lenguaje Java tiene predefinidas las siguientes secuencias de escape:

Secuencia	ASCII	Definición
\n	CR+LF	Ir al principio de la siguiente línea
\t	HT	Tabulador horizontal
\b	BS	Retroceso (<i>backspace</i>)
\r	CR	Retorno de carro sin avance de línea
\f	FF	Alimentación de página (sólo para impresora)
\'	'	Comilla simple
\"	"	Comilla doble
\\	\	Barra invertida (<i>backslash</i>)
\ddd		Carácter ASCII. Representación octal
\udddd		Carácter ASCII. Representación Unicode
\u0007	BEL	Alerta, pitido
\u000B	VT	Tabulador vertical (sólo para impresora)

Observe en el ejemplo anterior la secuencia de escape `\n` en la llamada al método `print`.

TIPOS DE DATOS

Recuerde las operaciones aritméticas que realizaba el programa *Aritmetica.java* que vimos en un capítulo anterior. Por ejemplo, una de las operaciones que realizábamos era la suma de dos valores:

```
dato1 = 20; dato2 = 10; resultado = dato1 + dato2;
```

Para que el compilador Java reconozca esta operación es necesario especificar previamente el tipo de cada uno de los operandos que intervienen en la misma, así como el tipo del resultado. Para ello, escribiremos una línea como la siguiente:

```
int dato1, dato2, resultado;
```

La declaración anterior le dice al compilador Java que *dato1*, *dato2* y *resultado* son de tipo entero (**int**).

Los tipos de datos en Java se clasifican en: tipos *primitivos* y tipos *referenciados*.

Tipos primitivos

Hay ocho tipos primitivos de datos que podemos clasificar en: tipos numéricos y el tipo **boolean**. A su vez, los tipos numéricos se clasifican en tipos enteros y tipos reales.

Tipos enteros: **byte**, **short**, **int**, **long** y **char**.

Tipos reales: **float** y **double**.

Cada tipo primitivo tiene un rango diferente de valores positivos y negativos, excepto el **boolean** que sólo tiene dos valores: **true** y **false**. El tipo de datos que se seleccione para declarar las variables de un determinado programa dependerá del rango y tipo de valores que vayan a almacenar cada una de ellas y de si éstos son enteros o fraccionarios.

Se les llama primitivos porque están integrados en el sistema y en realidad no son objetos, lo cual hace que su uso sea más eficiente. Más adelante veremos también que la biblioteca Java proporciona las clases: **Byte**, **Character**, **Short**, **Integer**, **Long**, **Float**, **Double** y **Boolean**, para encapsular cada uno de los tipos expuestos, proporcionando así una funcionalidad añadida para manipularlos.

byte

El tipo **byte** se utiliza para declarar datos enteros comprendidos entre -128 y $+127$. Un **byte** se define como un conjunto de 8 bits, independientemente de la plataforma en que se ejecute el *código byte* de Java. El siguiente ejemplo declara la variable *b* de tipo **byte** y le asigna el valor inicial 0. Es recomendable iniciar toda variable que se declare.

```
byte b = 0;
```

short

El tipo **short** se utiliza para declarar datos enteros comprendidos entre -32768 y $+32767$. Un valor **short** se define como un dato de 16 bits de longitud, independientemente de la plataforma en la que resida el *código byte* de Java. El siguiente ejemplo declara *i* y *j* como variables enteras de tipo **short**:

```
short i = 0, j = 0;
```

int

El tipo **int** se utiliza para declarar datos enteros comprendidos entre -2147483648 y $+2147483647$. Un valor **int** se define como un dato de 32 bits de longitud, independientemente de la plataforma en la que se ejecute el *código byte* de Java. El siguiente ejemplo declara e inicia tres variables *a*, *b* y *c*, de tipo **int**:

```
int a = 2000;
int b = -30;
int c = 0xF003; /* valor en hexadecimal */
```

En general, el uso de enteros de cualquier tipo produce un código compacto y rápido. Así mismo, podemos afirmar que la longitud de un **short** es siempre menor o igual que la longitud de un **int**.

long

El tipo **long** se utiliza para declarar datos enteros comprendidos entre los valores -9223372036854775808 y $+9223372036854775807$. Un valor **long** se define como un dato de 64 bits de longitud, independientemente de la plataforma en la que se ejecute el *código byte* de Java. El siguiente ejemplo declara e inicia las variables *a*, *b* y *c*, de tipo **long**:

```
long a = -1L; /* L indica que la constante -1 es long */
long b = 125;
```

```
long c = 0x1F00230F; /* valor en hexadecimal */
```

En general, podemos afirmar que la longitud de un **int** es menor o igual que la longitud de un **long**.

char

El tipo **char** es utilizado para declarar datos enteros en el rango `\u0000` a `\uFFFF` en *Unicode* (0 a 65535). Los valores 0 a 127 se corresponden con los caracteres ASCII del mismo código (ver los apéndices). El juego de caracteres ASCII conforman una parte muy pequeña del juego de caracteres *Unicode*.

```
char car = 0;
```

En Java para representar los caracteres se utiliza el código *Unicode*. Se trata de un código de 16 bits (esto es, cada carácter ocupa 2 bytes) con el único propósito de internacionalizar el lenguaje. El código *Unicode* actualmente representa los caracteres de la mayoría de los idiomas escritos conocidos en todo el mundo.

El siguiente ejemplo declara la variable *car* de tipo **char** a la que se le asigna el carácter 'a' como valor inicial (observe que hay una diferencia entre 'a' y *a*; *a* entre comillas simples es interpretada por el compilador Java como un valor, un carácter, y *a* sin comillas sería interpretada como una variable). Las cuatro declaraciones siguientes son idénticas:

```
char car = 'a';  
char car = 97;      /* la 'a' es el decimal 97 */  
char car = 0x0061;  /* la 'a' es el hexadecimal 0061 */  
char car = '\u0061'; /* la 'a' es el Unicode 0061 */
```

Un carácter es representado internamente por un entero, que puede ser expresado en decimal, hexadecimal u octal, como veremos más adelante.

float

El tipo **float** se utiliza para declarar un dato en coma flotante de 32 bits en el formato IEEE 754 (este formato utiliza 1 bit para el signo, 8 bits para el exponente y 24 para la mantisa). Los datos de tipo **float** almacenan valores con una precisión aproximada de 7 dígitos. Para especificar que una constante (un literal) es de tipo **float**, hay que añadir al final de su valor la letra 'f' o 'F'. El siguiente ejemplo declara las variables *a*, *b* y *c*, de tipo real de precisión simple:

```
float a = 3.14159F;  
float b = 2.2e-5F; /* 2.2e-5 = 2.2 por 10 elevado a 5 */  
float c = 2/3F; /* 0,6666667 */
```

double

El tipo **double** se utiliza para declarar un dato en coma flotante de 64 bits en el formato IEEE 754 (1 bit para el signo, 11 bits para el exponente y 52 para la mantisa). Los datos de tipo **double** almacenan valores con una precisión aproximada de 16 dígitos. Para especificar explícitamente que una constante (un literal) es de tipo **double**, hay que añadir al final de su valor la letra 'd' o 'D'; por omisión, una constante es considerada de tipo **double**. El siguiente ejemplo declara las variables *a*, *b* y *c*, de tipo real de precisión doble:

```
double a = 3.14159; /* una constante es double por omisión */
double b = 2.2e+5; /* 2.2e-5 = 2.2 por 10 elevado a 5 */
double c = 2/3D;
```

boolean

El tipo **boolean** se utiliza para indicar si el resultado de la evaluación de una expresión booleana es verdadero o falso. Los dos posibles valores de una expresión booleana son **true** y **false**. Los literales **true** y **false** son constantes definidas como palabras clave en el lenguaje Java. Por tanto, se pueden utilizar las palabras **true** y **false** como valores de retorno, en expresiones condicionales, en asignaciones y en comparaciones con otras variables booleanas.

El contenido de una variable booleana no se puede convertir a otros tipos, pero sí se puede convertir en una cadena de caracteres.

Tipos referenciados

Hay tres clases de tipos referenciados: clases, interfaces y *arrays*. Todos ellos serán objeto de estudio en capítulos posteriores.

LITERALES

Un literal es la expresión de un valor de un tipo primitivo, de un tipo **String** (cadena de caracteres) o la expresión **null** (valor nulo o desconocido). Por ejemplo, son literales: 5, 3.14, 'a', "hola" y **null**. En realidad son valores constantes.

Un literal en Java puede ser: un entero, un real, un valor booleano, un carácter, una cadena de caracteres y un valor nulo.

Literales enteros

El lenguaje Java permite especificar un literal entero en base 10, 8 y 16.

En general, el signo + es opcional si el valor es positivo y el signo - estará presente siempre que el valor sea negativo. Un literal entero es de tipo **int** a no ser que su valor absoluto sea mayor que el de un **int** o se especifique el sufijo *l* o *L*, en cuyo caso será **long**. Lo expuesto queda resumido en la línea siguiente:

```
([+]|-)?literal_entero([l]|L)?
```

Un *literal entero decimal* puede tener uno o más dígitos del 0 a 9, de los cuales el primero de ellos es distinto de 0. Por ejemplo:

```
4326           constante entera int
4326L          constante entera long
3426000000     constante entera long
```

Un *literal entero octal* puede tener uno o más dígitos del 0 a 7, precedidos por 0 (cero). Por ejemplo:

```
0326           constante entera int en base 8
```

Un *literal entero hexadecimal* puede tener uno o más dígitos del 0 a 9 y letras de la A a la F (en mayúsculas o en minúsculas) precedidos por 0x o 0X (*cero* seguido de *x*). Por ejemplo:

```
256           número decimal 256
0400          número decimal 256 expresado en octal
0x100         número decimal 256 expresado en hexadecimal
-0400         número decimal -256 expresado en octal
-0x100        número decimal -256 expresado en hexadecimal
```

Literales reales

Un literal real está formado por una *parte entera*, seguido por un *punto decimal*, y una *parte fraccionaria*. También se permite la notación científica, en cuyo caso se añade al valor una *e* o *E*, seguida por un exponente positivo o negativo.

```
([+]|-)?parte-entera.parte-fraccionaria([e|E])([+]|-)?exponente
```

donde *exponente* representa cero o más dígitos del 0 al 9 y *E* o *e* es el símbolo de exponente de la base 10 que puede ser positivo o negativo ($2E-5 = 2 \times 10^{-5}$). Si

la constante real es positiva no es necesario especificar el signo y si es negativa lleva el signo menos (-). Por ejemplo:

```
-17.24
17.244283
.008e3
27E-3
```

Una constante real tiene siempre tipo **double**, a no ser que se añada a la misma una *f* o *F*, en cuyo caso será de tipo **float**. Por ejemplo:

```
17.24F    constante real de tipo float
```

También se pueden utilizar los sufijos *d* o *D* para especificar explícitamente que se trata de una constante de tipo **double**. Por ejemplo:

```
17.24D    constante real de tipo double
```

Literales de un solo carácter

Los literales de un solo carácter son de tipo **char**. Este tipo de literales está formado por un único carácter encerrado entre *comillas simples*. Una secuencia de escape es considerada como un único carácter. Algunos ejemplos son:

```
' '      espacio en blanco
'x'      letra minúscula x
'\n'     retorno de carro más avance de línea
'\u0007' pitido
'\u001B' carácter ASCII Esc
```

El valor de una constante de un solo carácter es el valor que le corresponde en el juego de caracteres de la máquina.

Literales de cadenas de caracteres

Un literal de cadena de caracteres es una secuencia de caracteres encerrados entre *comillas dobles* (incluidas las secuencias de escape como `\`). Por ejemplo:

```
"Esto es una constante de caracteres"
"3.1415926"
"Paseo de Pereda 10, Santander"
""                /* cadena vacía */
"Lenguaje \"Java\"" /* produce: Lenguaje "Java" */
```

En el ejemplo siguiente el carácter `\n` fuerza a que la cadena “*O pulse Entrar*” se escriba en una nueva línea:

```
System.out.print("Escriba un número entre 1 y 5\n0 pulse Entrar");
```

Las cadenas de caracteres en Java son objetos de la clase **String** que estudiaremos más adelante. Esto es, cada vez que en un programa se utilice un literal de caracteres, Java crea de forma automática un objeto **String** con el valor del literal.

Las cadenas de caracteres se pueden concatenar (unir) empleando el operador `+`. Por ejemplo, la siguiente sentencia concatena las cadenas “*Distancia:* ”, *distancia*, y “*Km.*”.

```
System.out.println("Distancia: " + distancia + " Km.");
```

Si alguna de las expresiones no se corresponde con una cadena, como se supone que ocurre con *distancia*, Java la convierte de forma automática en una cadena de caracteres. Más adelante aprenderá el porqué de esto.

IDENTIFICADORES

Los identificadores son nombres dados a tipos, literales, variables, clases, interfaces, métodos, paquetes y sentencias de un programa. La sintaxis para formar un identificador es la siguiente:

```
{letra|_|$}[(letra|dígito|_|$)]...
```

lo cual indica que un identificador consta de uno o más caracteres (véase el apartado anterior “*Letras, dígitos y otros*”) y que el *primer carácter* debe ser una *letra*, el *carácter de subrayado* o el *carácter dólar* (\$). No pueden comenzar por un dígito ni pueden contener caracteres especiales (véase el apartado anterior “*Caracteres especiales*”).

Las letras pueden ser mayúsculas o minúsculas. Para Java una letra mayúscula es un carácter diferente a esa misma letra en minúscula. Por ejemplo, los identificadores *Suma*, *suma* y *SUMA* son diferentes.

Los identificadores pueden tener cualquier número de caracteres. Algunos ejemplos son:

```
Suma
Cálculo_Números_Primos
$ordenar
VisualizarDatos
```

PALABRAS CLAVE

Las palabras clave son identificadores predefinidos que tienen un significado especial para el compilador Java. Por lo tanto, un identificador definido por el usuario, no puede tener el mismo nombre que una palabra clave. El lenguaje Java, tiene las siguientes palabras clave:

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

Las palabras clave deben escribirse siempre en minúsculas, como están.

COMENTARIOS

Un comentario es un mensaje a cualquiera que lea el código fuente. Añadiendo comentarios se hace más fácil la comprensión de un programa. La finalidad de los comentarios es explicar el código fuente. Java soporta tres tipos de comentarios:

- *Comentario tradicional.* Un comentario tradicional empieza con los caracteres `/*` y finaliza con los caracteres `*/`. Estos comentarios pueden ocupar más de una línea, pero no pueden anidarse. Por ejemplo:

```
/*
 * La ejecución del programa comienza con el método main().
 * La llamada al constructor de la clase no tiene lugar a menos
 * que se cree un objeto del tipo 'CElementosJava'
 * en el método main().
 */
```

- *Comentario de una sola línea.* Este tipo de comentario comienza con una doble barra (`//`) y se extiende hasta el final de la línea. Por ejemplo:

```
// Agregar aquí el código de iniciación
```

- *Comentario de documentación.* Este tipo de comentario comienza con `/**` y termina con `*/`. Son comentarios especiales que *javadoc* utiliza para generar la

documentación acerca del programa, aunque también se pueden emplear de manera idéntica a los comentarios tradicionales.

```
/**
 * Punto de entrada principal para la aplicación.
 *
 * Parámetros:
 *   args: Matriz de parámetros pasados a la aplicación
 *         a través de la línea de órdenes.
 */
```

DECLARACIÓN DE CONSTANTES SIMBÓLICAS

Declarar una constante simbólica significa decirle al compilador Java el nombre de la constante y su valor. Esto se hace utilizando el calificador **final** y/o el **static**.

```
class CElementosJava
{
    final static int cte1 = 1;
    final static String cte2 = "Pulse una tecla para continuar";

    void Test()
    {
        final double cte3 = 3.1415926;
        // ...
    }
    // ...
}
```

Como se observa en el ejemplo anterior, declarar una constante simbólica supone anteponer el calificador **final**, o bien los calificadores **final** y **static**, al tipo y nombre de la constante, que será iniciada con el valor deseado. Distinguimos dos casos: que la constante esté definida en el cuerpo de la clase, fuera de todo método, como sucede con *cte1* y *cte2*, o que esté definida dentro de un método, como sucede con *cte3*. En el primer caso, la constante puede estar calificada, además de con **final**, con **static**; en este caso, sólo existirá una copia de la constante para todos los objetos que se declaren de esa clase (en nuestro caso, la clase es *CElementosJava*). Si no se especifica **static**, cada objeto incluiría su propia copia de la constante; es claro que esta forma de proceder no parece lógica por tratarse de la misma constante, razón por la que no se hace uso de ella. En el segundo caso no se puede utilizar **static**, la constante sólo es visible dentro del método, y sólo existe durante la ejecución del mismo; en este caso se dice que la constante es *local* al método. Una constante local no pueden ser declarada **static**.

Una vez que se haya declarado una constante, por definición, no se le puede asignar otro valor. Por ello, cuando se declara una constante debe ser iniciada con un valor. Por ejemplo, después de haber declarado *cte3* según se muestra en el ejemplo anterior, una sentencia como la siguiente daría lugar a un error:

```
cte3 = 3.14;
```

¿Por qué utilizar constantes?

Utilizando constantes es más fácil modificar un programa. Por ejemplo, supongamos que un programa utiliza N veces una constante de valor 3.14. Si hemos definido dicha constante como *final static double Pi = 3.14* y posteriormente necesitamos cambiar el valor de la misma a 3.1416, sólo tendremos que modificar una línea, la que define la constante. En cambio, si no hemos declarado *Pi*, sino que hemos utilizado el valor 3.14 directamente N veces, tendríamos que realizar N cambios.

DECLARACIÓN DE UNA VARIABLE

Una variable representa un espacio de memoria para almacenar un valor de un determinado tipo. El valor de una variable, a diferencia de una constante, puede cambiar durante la ejecución de un programa. Para utilizar una variable en un programa, primero hay que declararla. La declaración de una variable consiste en enunciar el nombre de la misma y asociarle un tipo:

```
tipo identificador[, identificador]...
```

En el ejemplo siguiente se declaran tres variables de tipo **short**, una variable de tipo **int**, y dos variables de tipo **String**:

```
class CElementosJava
{
    short día, mes, año;

    void Test()
    {
        int contador = 0;
        String Nombre = "", Apellidos = "";
        día = 20;
        Apellidos = "Ceballos";
        // ...
    }
    // ...
}
```

El tipo, primitivo o referenciado, determina los valores que puede tomar la variable así como las operaciones que con ella pueden realizarse. Los operadores serán expuestos un poco más adelante.

Por definición, una variable declarada dentro de un bloque, entendiendo por bloque el código encerrado entre los caracteres '{' y '}', es accesible directamente, esto es, sin un objeto, sólo dentro de ese bloque. Más adelante, cuando tratemos con objetos matizaremos el concepto de accesibilidad.

Según la definición anterior, las variables *día*, *mes* y *año* son accesibles desde todos los métodos no **static** de la clase *CElementosJava*. Estas variables, declaradas en el bloque de la clase pero fuera de cualquier otro bloque, se denominan *variables miembro* de la clase (atributos de la clase).

En cambio, las variables *contador*, *Nombre* y *Apellidos* han sido declaradas en el bloque de código correspondiente al cuerpo del método *Test*. Por lo tanto, aplicando la definición anterior, sólo serán accesibles en este bloque. En este caso se dice que dichas variables son *locales* al bloque donde han sido declaradas. Una variable local se crea cuando se ejecuta el bloque donde se declara y se destruye cuando finaliza la ejecución de dicho bloque; dicho de otra forma, una variable local se destruye cuando el flujo de ejecución sale fuera del ámbito de la variable. Una variable local no puede ser declarada **static**.

Iniciación de una variable

Las variables miembro de una clase son iniciadas por omisión por el compilador Java para cada objeto que se declare de la misma: las variables numéricas con 0, los caracteres con '\0' y las referencias a las cadenas de caracteres y el resto de las referencias a otros objetos con **null**. También pueden ser iniciadas explícitamente. En cambio, las variables locales no son iniciadas por el compilador Java. Por lo tanto, es nuestra obligación iniciarlas, de lo contrario el compilador visualizará un mensaje de error en todas las sentencias que hagan referencia a esas variables.

```
class CElementosJava
{
    short var1;
    void Test()
    {
        int var2;
        System.out.println(var2);    // error: variable no iniciada
        System.out.println(var1);    // correcto: var1 es igual a 0
    }
    // ...
}
```

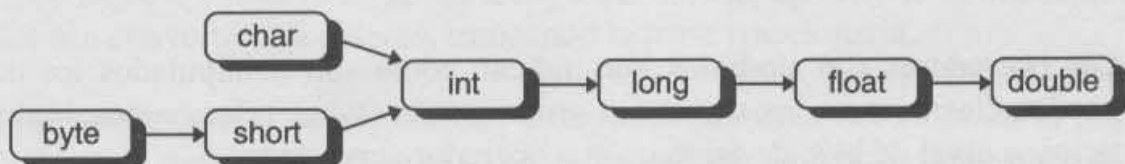
EXPRESIONES NUMÉRICAS

Una expresión es un conjunto de operandos unidos mediante operadores para especificar una operación determinada. Todas las expresiones cuando se evalúan retornan un valor. Por ejemplo:

```
a + 1
suma + c
cantidad * precio
7 * Math.sqrt(a) - b / 2      (sqrt indica raíz cuadrada)
```

CONVERSIÓN ENTRE TIPOS DE DATOS

Cuando Java tiene que evaluar una expresión en la que intervienen operandos de diferentes tipos, primero convierte, sólo para realizar las operaciones solicitadas, los valores de los operandos al tipo del operando cuya precisión sea más alta. Cuando se trate de una asignación, convierte el valor de la derecha al tipo de la variable de la izquierda siempre que no haya pérdida de información. En otro caso, Java exige que la conversión se realice explícitamente. La figura siguiente resume los tipos colocados de izquierda a derecha de menos a más precisos; las flechas indican las conversiones implícitas permitidas:



```
// Conversión implícita
byte bDato = 1; short sDato = 0; int iDato = 0; long lDato = 0;
float fDato = 0; double dDato = 0;

sDato = bDato;
iDato = sDato;
lDato = iDato;
fDato = lDato;
dDato = fDato + lDato - iDato * sDato / bDato;
System.out.println(dDato); // resultado: 1.0
```

Java permite una conversión explícita (conversión forzada) del tipo de una expresión mediante una construcción denominada *cast*, que tiene la forma:

(tipo) expresión

Cualquier valor de un tipo entero o real puede ser convertido a o desde cualquier tipo numérico. No se pueden realizar conversiones entre los tipos enteros o reales y el tipo **boolean**. Por ejemplo:

```
// Conversión explícita (cast)
byte bDato = 0; short sDato = 0; int iDato = 0; long lDato = 0;
float fDato = 0; double dDato = 2;

fDato = (float)dDato;
lDato = (long)fDato;
iDato = (int)lDato;
sDato = (short)iDato;
bDato = (byte)(sDato + iDato - lDato * fDato / dDato);
System.out.println(bDato); // resultado: 2
```

La expresión es convertida al tipo especificado si esa conversión está permitida; en otro caso, se obtendrá un error. La utilización apropiada de construcciones *cast* garantiza una evaluación consistente, pero siempre que se pueda, es mejor evitarla ya que suprime la verificación de tipo proporcionada por el compilador y por consiguiente puede conducir a resultados inesperados, o cuando menos, a una pérdida de precisión en el resultado. Por ejemplo:

```
float r;
r = (float)Math.sqrt(10); // el resultado se redondea perdiendo
                          // precisión ya que sqrt devuelve un
                          // valor de tipo double
```

OPERADORES

Los operadores son símbolos que indican cómo son manipulados los datos. Se pueden clasificar en los siguientes grupos: aritméticos, relacionales, lógicos, unitarios, a nivel de bits, de asignación y operador condicional.

Operadores aritméticos

Los operadores aritméticos los utilizamos para realizar operaciones matemáticas y son los siguientes:

Operador	Operación
+	<i>Suma.</i> Los operandos pueden ser enteros o reales.
-	<i>Resta.</i> Los operandos pueden ser enteros o reales.
*	<i>Multiplicación.</i> Los operandos pueden ser enteros o reales.
/	<i>División.</i> Los operandos pueden ser enteros o reales. Si ambos operandos son enteros el resultado es entero. En el resto de los casos el resultado es real.
%	<i>Módulo</i> o resto de una división entera. Los operandos tienen que ser enteros.

El siguiente ejemplo muestra cómo utilizar estos operadores. Como ya hemos venido diciendo, observe que primero se declaran las variables y después se realizan las operaciones deseadas con ellas.

```
int a = 10, b = 3, c;
float x = 2.0F, y;
y = x + a;          // El resultado es 12.0 de tipo float
c = a / b;          // El resultado es 3 de tipo int
c = a % b;          // El resultado es 1 de tipo int
y = a / b;          // El resultado es 3 de tipo int. Se
                    // convierte a float para asignarlo a y
c = (int)(x / y);    // El resultado es 0.6666667 de tipo float. Se
                    // convierte a int para asignarlo a c (c = 0)
```

Cuando en una operación aritmética los operandos son de diferentes tipos, ambos son convertidos al tipo del operando de precisión más alta. Por ejemplo, para realizar la suma $x+a$ el valor del entero a es convertido a **float**, tipo de x . No se modifica a , sino que su valor es convertido a **float** sólo para realizar la suma. Los tipos **short** y **byte** son convertidos de manera automática a **int**.

En una asignación, el resultado obtenido en una operación aritmética es convertido implícita o explícitamente al tipo de la variable que almacena dicho resultado (véase "Conversión entre tipos de datos"). Por ejemplo, del resultado de x/y sólo la parte entera es asignada a c , ya que c es de tipo **int**. Esto indica que los reales son convertidos a enteros, truncando la parte fraccionaria.

Un resultado real es redondeado independientemente del valor de la primera cifra decimal suprimida. Observe la operación x/y para x igual a 2 e y igual a 3. El resultado es 0.6666667 en lugar de 0.6666666.

Operadores de relación

Los operadores de relación o de comparación permiten evaluar la igualdad y la magnitud. El resultado de una operación de relación es un valor booleano **true** o **false**. Los operadores de relación son los siguientes:

Operador	Operación
<	¿Primer operando <i>menor que</i> el segundo?
>	¿Primer operando <i>mayor que</i> el segundo?
<=	¿Primer operando <i>menor o igual que</i> el segundo?
>=	¿Primer operando <i>mayor o igual que</i> el segundo?
!=	¿Primer operando <i>distinto que</i> el segundo?
==	¿Primer operando <i>igual que</i> el segundo?

Los operandos tiene que ser de un tipo primitivo. Por ejemplo:

```
int x = 10, y = 0;
boolean r;

r = x == y;    // r = false
r = x > y;     // r = true
r = x != y;    // r = true
```

Un operador de relación equivale a una pregunta relativa a cómo son dos operandos entre sí. Por ejemplo, la expresión `x == y` equivale a la pregunta ¿x es igual a y? Una respuesta sí equivale a un valor verdadero (**true**) y una respuesta no equivale a un valor falso (**false**).

Operadores lógicos

El resultado de una operación lógica (AND, OR, XOR y NOT) es un valor booleano verdadero o falso (**true** o **false**). Las expresiones que dan como resultado valores booleanos (véanse los operadores de relación) pueden combinarse para formar expresiones *booleanas* utilizando los operadores lógicos indicados a continuación. Los operandos deben ser expresiones que den un resultado **boolean**.

Operador	Operación
&& o &	<i>AND</i> . Da como resultado true si al evaluar cada uno de los operandos el resultado es true . Si uno de ellos es false , el resultado es false . Si se utiliza && (no &) y el primer operando es false , el segundo operando no es evaluado.
o	<i>OR</i> . El resultado es false si al evaluar cada uno de los operandos el resultado es false . Si uno de ellos es true , el resultado es true . Si se utiliza (no) y el primer operando es true , el segundo operando no es evaluado (el carácter es el ASCII 124).
!	<i>NOT</i> . El resultado de aplicar este operador es false si al evaluar su operando el resultado es true , y true en caso contrario.
^	<i>XOR</i> . Da como resultado true si al evaluar cada uno de los operandos el resultado de uno es true y el del otro false ; en otro caso el resultado es false .

El resultado de una operación lógica es de tipo **boolean**. Por ejemplo:

```
int p = 10, q = 0;
boolean r;

r = p != 0 && q != 0;    // r = false
```

```

r = p != 0 || q > 0;    // r = true
r = q < p && p <= 10;   // r = true
r = !r;                 // si r = true, entonces r = false

```

Operadores unitarios

Los operadores unitarios se aplican a un solo operando y son los siguientes: `!`, `-`, `~`, `++` y `--`. El operador `!` ya lo hemos visto y los operadores `++` y `--` los veremos más adelante.

Operador	Operación
<code>~</code>	Complemento a 1 (cambiar ceros por unos y unos por ceros). El carácter <code>~</code> es el ASCII 126. El operando debe de ser de un tipo primitivo entero.
<code>-</code>	Cambia de signo al operando (esto es, se calcula el complemento a dos que es el complemento a 1 más 1). El operando puede ser de un tipo primitivo entero o real.

El siguiente ejemplo muestra cómo utilizar estos operadores:

```

int a = 2, b = 0, c = 0;

c = -a;    // resultado c = -2
c = ~b;    // resultado c = -1

```

Operadores a nivel de bits

Estos operadores permiten realizar con sus operandos las operaciones AND, OR, XOR y desplazamientos, bit por bit. Los operandos tienen que ser enteros.

Operador	Operación
<code>&</code>	Operación AND a nivel de bits.
<code> </code>	Operación OR a nivel de bits (carácter ASCII 124).
<code>^</code>	Operación XOR a nivel de bits.
<code><<</code>	Desplazamiento a la izquierda rellenando con ceros por la derecha.
<code>>></code>	Desplazamiento a la derecha rellenando con el bit de signo por la izquierda.
<code>>>></code>	Desplazamiento a la derecha rellenando con ceros por la izquierda.

Los operandos tienen que ser de un tipo primitivo entero.

```

int a = 255, r = 0, m = 32;

r = a & 017; // r=15. Pone a cero todos los bits de a
              // excepto los 4 bits de menor peso.
r = r | m;   // r=47. Pone a 1 todos los bits de r que
              // estén a 1 en m.
r = a & ~07; // r=248. Pone a 0 los 3 bits de menor peso de a.
r = a >> 7;  // r=1. Desplazamiento de 7 bits a la derecha.
r = m << 1;  // r=64. Equivale a r = m * 2
r = m >> 1;  // r=16. Equivale a r = m / 2

```

Operadores de asignación

El resultado de una operación de asignación es el valor almacenado en el operando izquierdo, lógicamente después de que la asignación se ha realizado. El valor que se asigna es convertido implícita o explícitamente al tipo del operando de la izquierda (véase el apartado “Conversión entre tipos de datos”). Incluimos aquí los operadores de incremento y decremento porque implícitamente estos operadores realizan una asignación sobre su operando.

Operador	Operación
++	Incremento.
--	Decremento.
=	Asignación simple.
*=	Multiplicación más asignación.
/=	División más asignación.
%=	Módulo más asignación.
+=	Suma más asignación.
-=	Resta más asignación.
<<=	Desplazamiento a izquierdas más asignación.
>>=	Desplazamiento a derechas más asignación.
>>>	Desplazamiento a derechas más asignación rellenando con ceros.
&=	Operación AND sobre bits más asignación.
=	Operación OR sobre bits más asignación.
^=	Operación XOR sobre bits más asignación.

Los operandos tienen que ser de un tipo primitivo. A continuación se muestran algunos ejemplos con estos operadores.

```

int x = 0, n = 10, i = 1;
n++;           // Incrementa el valor de n en 1.

```



```

++n;           // Incrementa el valor de n en 1.
x = ++n;       // Incrementa n en 1 y asigna el resultado a x.
x = n++;       // Asigna el valor de n a x y después
               // incrementa n en 1.
i += 2;        // Realiza la operación i = i + 2.
x *= n - 3;    // Realiza la operación x = x * (n-3) y no
               // x = x * n - 3.
n >>= 1;       // Realiza la operación n = n >> 1 la cual desplaza
               // el contenido de n 1 bit a la derecha.

```

El operador de incremento incrementa su operando independientemente de que se utilice como sufijo o como prefijo; esto es, $n++$ y $++n$ producen el mismo resultado. Ídem para el operador de decremento.

Ahora bien, cuando el resultado de una operación de incremento se asigna a una variable, como se puede observar en $x = ++n$ y $x = n++$, si el operador de incremento se utiliza como prefijo primero se realiza la operación de incremento y después la asignación; y si se utiliza como sufijo, primero se realiza la operación de asignación y después la de incremento. Ídem para el operador de decremento.

Operador condicional

El operador condicional ($?:$), llamado también operador ternario, se utiliza en expresiones condicionales, que tienen la forma siguiente:

operando1 ? operando2: operando3

La expresión *operando1* debe ser una expresión booleana. La ejecución se realiza de la siguiente forma:

- Si el resultado de la evaluación de *operando1* es **true**, el resultado de la expresión condicional es *operando2*.
- Si el resultado de la evaluación de *operando1* es **false**, el resultado de la expresión condicional es *operando3*.

El siguiente ejemplo asigna a *mayor* el resultado de $(a > b) ? a : b$, que será *a* si *a* es mayor que *b* y *b* si *a* no es mayor que *b*.

```

double a = 10.2, b = 20.5, mayor = 0;
mayor = (a > b) ? a : b;

```

PRIORIDAD Y ORDEN DE EVALUACIÓN

La tabla que se presenta a continuación, resume las reglas de prioridad y asociatividad de todos los operadores. Las líneas se han colocado de mayor a menor prioridad. Los operadores escritos sobre una misma línea tienen la misma prioridad.

Una expresión entre paréntesis, siempre se evalúa primero. Los paréntesis tienen mayor prioridad y son evaluados de más internos a más externos.

Operador	Asociatividad
() [] .	izquierda a derecha
- ~ ! ++ --	derecha a izquierda
new (tipo)expresión	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >> >>>	izquierda a derecha
< <= > >= instanceof	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
	izquierda a derecha
&&	izquierda a derecha
	izquierda a derecha
?:	derecha a izquierda
= *= /= %= += -= <<= >>= >>>= &= = ^=	derecha a izquierda

En Java, todos los operadores binarios excepto los de asignación son evaluados de izquierda a derecha. En el siguiente ejemplo, primero se asigna *z* a *y* y a continuación *y* a *x*.

```
int x = 0, y = 0, z = 15;
x = y = z;      // resultado x = y = z = 15
```

EJERCICIOS RESUELTOS

La siguiente aplicación utiliza objetos de una clase *CEcuacion* para evaluar ecuaciones de la forma:

$$ax^3 + bx^2 + cx + d$$