Introducción a la programación en Java

por Pablo Ezequiel Jasinski

Antes de comenzar a programar en Java se deben conocer las bases del lenguaje. Al ser muy parecido a C y C++, no presenta grandes inconvenientes, ya que es probable que toda persona que sepa programar haya empezado con alguno de estos lenguajes.

Gestión de memoria en Java

La memoria juega un papel importante en cualquier lenguaje de programación, ya que para que un programa pueda ejecutarse en un sistema operativo, es necesario que exista una interfaz, más bien conocida como cargador (*linker*), que se encargue de suministrar al programa un bloque de memoria sobre el cual se vaya a ejecutar. Un óptimo manejo de la misma hará que un programa pueda ser más o menos eficaz cuando se ejecute. Es por ello que cada lenguaje utiliza algún sistema para gestionar la misma, y por lo general, es el programador quien debe estar atento de realizar esta gestión para no sobrecargar la misma. Uno de los aspectos o características que incorporó el lenguaje Java es la gestión dinámica de memoria, en donde, desliga completamente al programador de esta tarea, administrando la memoria automáticamente. Pero que Java desligue al programador en tareas de gestión de memoria, no significa que el mismo no deba conocer el funcionamiento de ésta gestión, ya que éstos conocimientos son útiles para la mejor comprensión de otros temas relacionados con la programación Orientada a Objetos.

Java trabaja con tres zonas de memoria: la zona de datos, La pila (*stack*) y el montículo (*heap*).

La zona de datos

En esta zona de memoria se guarda las instrucciones del programa ejecutable en código de máquina, las constantes, el código correspondiente a los métodos, y a las clases.
El tamaño de ésta zona en memoria se establece en tiempo de compilación, por lo cual al compilador le es necesario saber de antemano el tamaño de cada elemento que vaya a guardar, asegurándose de que éstos no cambien en tiempo de ejecución, ya que esta zona de memoria es fija, y no variará a lo largo de la ejecución del programa.

La pila (stack)

Con el surgimiento de lenguajes con estructura de bloques, surgió la necesidad de un manejo de memoria más flexible y eficiente, de manera que ésta pueda atender las necesidades en cuanto a demanda de memoria durante la ejecución del programa.

La pila funciona de manera en que lo primero que se guarda en ella será lo último en ser eliminado, y lo último que se guarda será lo primero en ser eliminado. Para entender el concepto se utiliza el acrónimo LIFO (*last in first out*), que hace referencia a lo explicado.

La analogía más común en este tema refiere al de una pila de platos. Para sacar platos se hará partiendo del primero, que fue el último en ser colocado en la pila, hasta el último, que fue el primero en ser colocado en la pila.

La pila cuenta con una cantidad limitada de memoria, y en ella se guardan variables locales, variables de referencia, parámetros y valores de retorno, resultados parciales, y el control de la invocación y retorno de métodos.

El montículo (heap)

El montículo o heap, a diferencia de la pila, es un espacio de memoria dinámico en donde se guardarán variables de instancia y objetos. El encargado de administrar este espacio de memoria es el Garbage Collector o recolector de basura, que libera al programador de estas tareas, a diferencia de lenguajes como C, en donde ésta gestión la realizaba el programador.

Atributos

Se habla de atributos de una clase, y en un lenguaje de programación Orientado a Objetos refieren a espacios de memoria donde se pueden almacenar datos. En otro tipo de paradigmas como en el estructurado estos atributos son llamados variables. Los mismos están conformados por un nombre que sirve para identificarlos dentro del programa y pueden estar asociados a un valor. Así mismo este valor está asociado a un tipo de dato, por lo que un atributo puede ser, por ejemplo, de tipo entero, lo que significa que en el mismo podrán guardarse datos solo de tipo numérico y enteros, o de tipo cadena, en donde se podrán guardar cadenas de caracteres, etc.

Los atributos pueden ser de longitud fija o variable. En el caso de atributos de longitud fija, se definen indicándole la cantidad de datos que pueden almacenar, y ésta cantidad no variará a lo largo de la ejecución del programa. Un ejemplo de atributos de longitud fija pueden ser los arreglos de datos (*array*). De la misma manera existen atributos de longitud variable, en donde como lo indica su definición, la cantidad de datos que pueden almacenar puede variar en la ejecución del programa. Un ejemplo de estos tipos de atributos son las colecciones de datos.

DEFINICION DE ATRIBUTOS

Tipos de datos primitivos

En POO se pueden distinguir dos clasificaciones en tipos de datos, los que se deben crear como objetos, y los tipos primitivos. Los tipos primitivos son tipos de datos sencillos y pequeños, con un tamaño conocido, los cuales son colocados en la pila, para que el programa resulte más eficiente.

Tipo de dato	Representación	Tamaño	Rango de valores	Valor por defecto	Clase asociada
boolean	Dato lógico	8 bits	-	false	Boolean
char	Carácter Unicode	16 bits	\u0000 a \uFFFF	\u0000	Character
byte	Número entero con signo	8 bits	-128 a 127	0	Byte
short	Número entero con signo	16 bits	-2^{15} a $+2^{15}-1$	0	Short
int	Número entero con signo	32 bits	-2^{31} a $+2^{31}-1$	0	Integer

long	Número entero con signo	64 bits	-2^{63} a $+2^{63}-1$	0	Long
float	Número en coma flotante de precisión simple Norma IEEE 754	32 bits	± 3.4x10-38 a ± 3.4x1038	0.0	Float
double	Número en coma flotante de precisión doble Norma IEEE 754	64 bits	± 1.8x10-308 a ± 1.8x10308	0.0	Double
void	-	-	-	-	Void

Observación: *Cuando se asigna un número decimal a un tipo de dato float, seguido del valor, se debe escribir la letra f para indicar que dicho valor es de tipo coma flotante. Ejemplo:* ***float*** *decimal = 10.5f; En casos que se desea guardar valores tipo 0.xxx Java permite la omisión del 0 pudiendo escribir .xxx. Ejemplo:* ***float*** *decimal = .5f;*

Tipos de referencia

Además de los tipos de datos primitivos, existen otros tipos de datos denominados de referencia. Estos tipos de datos están ligados al concepto de objetos, ya que justamente un tipo de "referencia", hace referencia a un objeto. Por lo tanto estos tipos serán referencias a objetos, a interfaces y a arreglos. Al referenciar a estos elementos, lo que se tiene en atributos de estos tipos, es en realidad, una posición de memoria que apunta a dichos elementos, y por lo tanto, si se crean dos atributos del mismo tipo, con algún tipo de contenido, y se asigna el primer atributo al segundo atributo, no se copiará un valor, sino que se pasará una dirección de memoria, por lo que, si el segundo atributo es modificado, evidentemente esos cambios se verán reflejados también en el primero.

Una referencia es por tanto un puntero a un objeto, pero a diferencia de otros lenguajes, el programador no obtiene directamente la dirección de memoria a la cual apunta la referencia, sino que es Java quien se encargará de obtener este objeto mediante esa dirección.

Para cada uno de los tipos primitivos, existe una clase asociada, que representa a estos tipos como un objeto. Por ejemplo para un tipo primitivo int existirá un tipo de referencia Int, para un tipo primitivo float existirá un tipo de referencia Float, etc.

La ventaja de tratar a un tipo primitivo como un objeto, es que mediante el objeto se podrá acceder a métodos, previamente definidos en cada clase de cada uno de estos tipos, los cuales pueden ser muy útiles, ya que mediante estos, se podrá, por ejemplo, convertir un tipo de dato entero en un tipo de dato String.

El tipo String

Java cuenta con una clase llamada String que da soporte especial a cadenas. En lenguajes como C, para crear una cadena de caracteres, se tenía que hacer un arreglo de elementos de tipo char, y cuando se asignaba una cadena de caracteres a esa variable, básicamente, se almacenaba cada carácter en una posición del arreglo. De hecho en Java también es posible hacer eso, pero como al trabajar de esa manera, se debe trabajar con un arreglo, lo cual tiene sus complicaciones, es mucho más sencillo hacerlo mediante el tipo String que ofrece Java.

Para guardar una cadena en un atributo de tipo String solo se deberá asignar mediante un signo = la cadena que se desee entre comillas dobles, y para concatenar cadenas se utilizará el operador "+".

El tipo de dato String no es un tipo primitivo, ya que al encerrar una cadena de caracteres entre comillas dobles, lo que Java hace realmente es crear un objeto de tipo String. Pero tampoco se puede decir que funciona exactamente como un tipo de referencia, ya que, en realidad, cuando se declara un tipo String se la declara como constante, lo que quiere decir, que una vez indicado el valor de la cadena, no podrá modificarse. Esto hará que al pasar este atributo como referencia por parámetro a algún método, y éste parámetro se modifique, no se observe cambio alguno en la cadena original (ver parámetros por referencia en unidad 4). Pero si se define un atributo de tipo String, se le asigna una cadena, y por ejemplo, seguido de eso se le asigna otra cadena a ese mismo atributo, si se verán los cambios, y esto se debe a que cuando se altera un valor de un atributo de tipo String, Java reserva un nuevo bloque de memoria para este nuevo valor asignado, dejando al anterior inservible.

Por lo tanto, si bien este tipo de dato no es un tipo de dato primitivo, debido al trato especial que le da Java, se tenderá a pensar en el cómo este tipo.

Conversiones de tipos

A menudo es necesario realizar conversiones de tipos de datos según el problema que se presente. Java realiza esta conversión automáticamente siempre que pueda, por ejemplo un valor de tipo byte podrá ser asignado a un valor de tipo int, y el resultado será un tipo int de 32 bits. Así mismo un valor de tipo int puede ser asignado a un valor de tipo long y el resultado será de tipo long con una longitud de 64 bits. Por lo cual, cuando un tipo de dato más chico se asigna a un tipo de dato más grande, Java realiza la conversión automáticamente, pero cuando el caso es al revés, no sucede esto, y si no se realiza una conversión, Java acusará error.

Por lo tanto, en éste último caso se deberá realizar una conversión, también conocida como casting. El casting es la conversión explicita de un tipo de dato a otro, y para realizar este tipo de conversión se debe anteponer al tipo origen, entre paréntesis, el nombre del tipo al que se lo desea convertir.

Ejemplo:

```
1   byte numero1;
2   int numero2 = 70;
3   numero1 = (byte) numero2;
```

Lo mismo sucederá entre tipos de datos int y float. En el caso que se asigne un valor de tipo int a un atributo de tipo float, Java hará la conversión automáticamente, pero si el caso es al revés, se debe hacer un casting.

Ejemplo:

```
1   int numero1;
2   float numero2 = 7.35f;
3   numero1 = (float) numero2;
```

Es posible, también, convertir tipos numéricos a cadenas y viceversa. Para realizar la conversión de un número a una cadena, se debe utilizar el tipo String, y utilizar un método llamado valueOf, en donde se le indica el valor a convertir, y este es devuelto como una cadena.

Ejemplo:

```
1   String numeroEnteroConvertido;
2   String numeroDecimalConvertido;
3   int entero = 77;
4   float numero2 = 4.2f;
5
6   numeroEnteroConvertido = String.valueOf(entero);
7   numeroDecimalConvertido = String.valueOf(decimal);
```

Otra manera de realizar esto es utilizando el método toString, del tipo de referencia al que se quiere convertir.

Ejemplo:

```
1   String numeroEnteroConvertido;
2   String numeroDecimalConvertido;
3   int entero = 77;
4   float decimal = 4.2f;
5
6   numeroEnteroConvertido = Integer.toString(entero);
7   numeroDecimalConvertido = Float.toString(decimal);
```

Para hacer la conversión inversa, o sea, de una cadena a un número, se debe utilizar el tipo parse*Tipo,* del tipo de referencia al que se quiere convertir.

Ejemplo:

```
1   String numero = "5";
2   int entero;
3   float decimal;
4
5   entero = Integer.parseInt(numero);
6   decimal = Float.parseFloat(numero);
```

Observación: *Cuando se realiza una conversión de una cadena a un número, se debe tener la certeza que en la cadena se encuentra un número válido, ya que si en ella se encontraran otro tipo de caracteres, Java acusaría error.*

Constantes

Las constantes a diferencia de los atributos, son valores que no se modifican a lo largo de todo el programa. Al definir una constante se debe indicar el valor que contendrá la misma, y luego se podrá consultar el valor de esta pero nunca modificar.

Para definir una constante se deben indicar en primera instancia dos palabras claves: static y final, luego el tipo del que será la misma y finalmente el nombre o identificador de la misma. La forma de la declaración de una constante es la siguiente:

```
static final tipo NOMBRE_DE_LA_CONSTANTE;
```

Ejemplos:

```
1   static final byte DIAS_SEMANA = 7;
2   static final float IVA = .21f;
3   static final String LENGUAJE = "Java";
```

Operadores

Los operadores sirven para realizar operaciones entre datos u operandos de tipos primitivos, lo cual dará como resultado un dato, también de tipo primitivo. Por ejemplo los operadores aritméticos sirven para realizar operaciones entre tipos de datos numéricos, y el resultado de dicha operación será también un tipo de dato numérico.

Los operadores se pueden agrupar en distintas clasificaciones. A continuación se expone una lista de los operadores que dispone Java, los cuales son muy similares a los de C y C++.

Operadores aritméticos

Sirven para realizar operaciones numéricas para tipos de datos numéricos enteros y reales.

Operador	Descripción	Ejemplo	Resultado
+	Suma	4 + 5	9
-	Resta	5.3 – 1.2	4.1
*	Producto	1.5 * 2.7	4.05
/	División	0.5 / 0.25 14 / 6	2.0 2
%	Resto de la división entera	20 % 7	6

Operadores de asignación

El operador de asignación, básicamente, asigna el valor del término de la derecha del operador, al término de la izquierda del mismo. En combinación con operadores aritméticos se pueden generar distintas funcionalidades.

Operador	Descripción	Ejemplo	Equivalencia
=	Asignación	a = b	a = b
+=	Suma combinada	a += b	a = a + b
-=	Resta combinada	a -= b	a = a - b
*=	Producto combinado	a *= b	a = a * b
/=	División combinada	a /= b	a = a / b
%=	Resto combinado	a %= b	a = a % b

Operadores unarios

Estos operadores sirven para cambiar de signo al operando.

Operador	Descripción	Ejemplo	Resultado
+	Operador unario de cambio de signo	+4	4
-	Operador unario de cambio de signo	-4	-4

Operadores relacionales

Sirven para comparar dos valores del mismo tipo, y como resultado se obtiene un tipo de dato lógico o booleano. Para la comparación de tipos lógicos, solo son válidos los operadores "==" y "!=".

Operador	Descripción	Ejemplo	Resultado
==	Igual que	'B' == 'b'	false
¡=	Distinto que	true ¡= false	true
<	Menor que	5 < 2	false
>	Mayor que	7.2 > 10.5	false
<=	Menor o igual que	'a' <= 'c'	true
>=	Mayor o igual que	5 >= 5	true

Operadores incrementales

Estos operandos sirven para incrementar en una unidad el valor de la variable en la que se la aplique. Se puede utilizar en tipos de datos enteros y de tipo carácter. Depende de qué lado del operando se ubique el operador, éste tendrá un comportamiento diferente.

Operador	Descripción	Ejemplo	Resultado
++	Se utiliza el atributo y luego se incrementa el valor	a = 5; 6 == a++;	5 false
	Se incrementa el valor del atributo y luego se la utiliza	a = 5; 6 == ++a;	5 true
--	Se utiliza el atributo y luego se decrementa el valor	a = 5; 4 == a--;	5 false
	Se decrementa el valor del atributo y luego se la utiliza	a = 5; 4 == --a;	5 true

Para entender mejor el concepto, se desarrollarán los ejemplos expuestos.

En el primer ejemplo se tiene que:

```
a = 5     // a vale 5
6 == a++ // el resultado de la comparación entre 6 y a++ es falso
```

En la primer línea se asigna el valor 5 a un atributo **a**, mientras que en la segunda se hace una comparación, la cual da por resultado falso, porque al tener el operador "++" delante del operando, primero hace la comparación y luego de haber hecho la comparación incrementa en 1 el valor de **a**.

De forma inversa actúa el operador "++" en el segundo ejemplo:

```
a = 5     // a vale 5
6 == ++a // el resultado de la comparación entre 6 y a++ es verdadero
```

Al igual que en el ejemplo anterior, en la primera línea se asigna el valor 5 al atributo **a**, y luego, en la segunda línea, primero se incrementa en 1 la variable **a**, por lo cual ahora pasará a valer 6, y luego se hace la comparación, la que finalmente da como resultado verdadero.

De esta misma manera actúan los operadores "--", solo que en vez de incrementar el valor del atributo en 1, lo decrementan, y dependiendo de la posición en que se encuentre el operando respecto al operador, actuará de la misma manera en la que se expuso en los ejemplos anteriores.

Operadores lógicos

Los operadores lógicos sirven para operar sobre tipos de datos lógicos o booleano, el resultado será del mismo tipo.

Operador	Descripción	Ejemplo	Resultado
!	Negación – NOT	!true !(10<5)	false true
|	Suma lógica – OR	true | false (5>8) | ('a' == 'a')	true true
^	Suma lógica exclusiva - XOR	true ^ false true ^ (7.3 < 10.1)	true true
&	Producto lógico – AND	true & false ('c' == 'c') & true	false true
||	Suma lógica con cortocircuito	true || false (5==5) || (3<1)	true true
&&	Producto lógico con cortocircuito	true && false (5>1) && (true!=false)	false true

La diferencia entre un operador sin cortocircuito y con cortocircuito radica en que el primero, hará todas las comparaciones, en cambio el operador con cortocircuito dejará de hacer comparaciones innecesarias cuando no haga falta.

Por ejemplo:

```
false &  true   // false
false && true   // false

true |  false   // true
true || false   // true
```

Las líneas 1 y 2 producen el mismo resultado, en este tipo de operación, se asume que si un valor resulta ser falso, el resultado será falso independientemente del resto de los valores que se encuentren, por lo que no requeriría de continuar la comprobación. En el caso de la primera línea, se realiza toda la comparación sin tener en cuenta esto, en cambio en la segunda línea se cuenta con un operador con cortocircuito, por lo cual, al detectar que el primer valor es falso, no se realizan más comparaciones y se devuelve como resultado de la operación el valor falso.

Lo mismo ocurre con el operador de suma lógica, si uno de los dos operandos es verdadero, entonces el resultado de la operación será verdadero, por lo tanto, si el primer operando posee valor verdadero, no haría falta seguir haciendo comparaciones, ya que se sabe que el resultado será verdadero independientemente del resto de operadores que se encuentren en la expresión. Entonces, de la misma manera, el operador sin cortocircuito, hará toda la comprobación, mientras que el operador con cortocircuito, dejará de hacer comprobaciones al detectar un valor verdadero, y dará como resultado el valor verdadero.

Operadores de bit

Sirven para realizar operaciones en binario, a excepción del operador de negación. Los operandos pueden ser de tipo entero o carácter y el resultado será de tipo entero. Estas operaciones se hacen

Operador	Descripción	Ejemplo	Resultado
~	Complemento binario	~7	-8
|	Suma lógica binaria	24|10	26
^	Suma lógica exclusiva binaria	14 ^ 7	9
&	Producto lógico binario	12 & 6	4
<<	Desplazamiento de bits hacia la izquierda	15 << 2	60
>>	Desplazamiento de bits hacia la derecha	53 >> 2	13
>>>	Desplazamiento de bits hacia la derecha sin signo	84 >>> 2	21

Para entender mejor estos operadores se detallarán los mismos mediante el desarrollo de ejemplos.

- La negación o complemento binario (~) invierte los bits del número, sustituyendo los ceros por unos y los unos por ceros, de ésta manera se obtiene el complemento del número, que la computadora usa para representar números negativos.

  Ejemplo:

  El número 7 en binario es 0111_2, si se aplica ~7 se obtendrá 1000_2, que en el sistema complemento a dos resulta ser -8. Para recordar éste operador se puede asumir que el resultado será –(x +1).

- La suma lógica binaria (|) consiste en tomar los dos números en binario, y realizar la operación OR bit a bit, esto quiere decir que si por lo menos uno de los dos bits es 1, entonces el resultado será 1.

  Ejemplo:

  Sean m = 24 (11000_2) y n = 10 (01010_2), m | n resultará 26.

$$\begin{array}{r} 11000 \\ 01010 \\ \hline 11010 \end{array} \;|$$

- La suma lógica exclusiva binaria (^) realiza la operación XOR bit a bit, lo cual implica que si los dos bits son iguales entonces el bit resultante será 0, mientras que si los dos bits son diferentes el bit resultante será 1.

  Ejemplo:

  Sean m = 14 (1110_2) y n = 7 (0111_2), m ^ n resultará 9.

  $$\begin{array}{r} 1110 \\ 0111 \\ \hline 1001 \end{array} \; ^{\wedge}$$

- El producto lógico binario (&) realiza la operación AND bit a bit, lo cual implica que si dos bits son 1 el resultado será 1, mientras que de otra forma el resultado será 0.

  Ejemplo:

  Sean m = 12 (1100_2) y n = 6 (0110_2), m & n resultará 4.

  $$\begin{array}{r} 1100 \\ 0110 \\ \hline 0100 \end{array} \; \&$$

Desplazamiento de bits

Los operadores de desplazamiento de bits suelen utilizarse para llevar a cabo operaciones muy rápidas de multiplicación y de división de enteros. Un desplazamiento a la izquierda equivale a una multiplicación por 2 y un desplazamiento a la derecha a una división por 2. A diferencia de C y C++, Java siempre conserva el bit de signo (el bit izquierdo) después de hacer un desplazamiento. Este tipo de desplazamiento se conoce como desplazamiento aritmético o desplazamiento de extensión de signo. Estas operaciones se realizan tomando 32 bits como longitud de los operandos. Java convierte a tipo int los tipo short y byte antes de realizar el desplazamiento de bits. Esto significa que el tipo byte tendrá 32 bits en lugar de 8. Luego de ejecutar el desplazamiento trunca el dato para convertir al tipo original.

- El desplazamiento de bits hacia la izquierda (<<), funciona desplazando a la izquierda los bits del primer operando tantas veces como lo indica el segundo operando. Por la derecha siempre entrará el valor 0. Esto es lo mismo que multiplicar el primer operando por 2 elevado al segundo operando. Este operador tiene en cuenta el signo.

  Ejemplo:

  Sean m = 15 (1111_2) y n = 2 (10_2), m << n resultará 111100_2 que en sistema decimal es 60 y n << m resultará 10000000000000000_2 que en sistema decimal es 65536. Si a m se le aplica signo negativo, el resultado tendrá signo negativo, o sea -60.

- El desplazamiento de bits hacia la derecha con signo (>>), funciona desplazando a la derecha los bits del primer operando tantas veces como indica el segundo operando. Por la izquierda entra siempre el bit más significativo anterior. Se debe tener en cuenta la cantidad de bits del tipo de dato sobre el cual se está trabajando, por ejemplo, un número entero es representado con 32 bits. Esto es lo mismo que dividir el primer operando por 2 elevado al segundo operando. Este operador tiene en cuenta el signo.

  Ejemplo:

  Sean m = 53 (110101_2) y n = 2 (10_2), m >> n resultará 001101_2 que en sistema decimal es 13.

Tenga en cuenta que si se está trabajando con un número entero la representación del mismo será:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011\ 0101_2$ por lo cual por la izquierda entrarán siempre ceros.

En el caso de m = -1 y n = 2, si se realiza m >> n el resultado será -1, y sea n cualquier otro valor el resultado será siendo 1. Esto se debe a que los números negativos se representan en complemento a dos esto quiere decir que el número 1 será representado de la siguiente manera:

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$ por lo tanto, sea cual sea el número que se desplace hacia la derecha, el número que ingresa por la izquierda será siempre 1, por lo que dará siempre el mismo resultado.

- El desplazamiento de bits hacia la derecha sin signo (>>>), funciona desplazando a la derecha los bits del primer operando tantas veces como indica el segundo operando sin tomar en cuenta el signo. Por la izquierda entra siempre entrará un cero.

  Ejemplo:

  Sean m = 84 (1010100_2) y n = 2, si se realiza m >>> n el resultado será 10101_2 que en sistema decimal es 21.

  Si m = -1 y n = 2, al realizarse m >>> n el resultado será 1073741823 ya que, como se mencionó anteriormente, los números negativos se representan en complemento a dos.

  $-1_{10} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$

  Por lo tanto $-1_{10} >>> 2_{10} = 0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 1073741823_{10}$

¿Tiene realmente algún sentido utilizar operadores a nivel bit?

Esto dependerá del problema que se quiera resolver, evidentemente estos operadores se usan en ciertos casos, de otra manera no tendría sentido su existencia en el lenguaje, ya que los creadores de un lenguaje no agregan características solo por fines teóricos.

Ejemplos de usos de los operadores a nivel bit son:

- **Programación de bajo nivel:** Algunos lenguajes, denominados de nivel medio, ofrecen la posibilidad de trabajar en bajo nivel, e incluir código ensamblador. El código ensamblador trabaja en binario, por lo cual, aquí estos operadores son sumamente útiles.
- **Programación de microcontroladores:** Para programar microcontroladores, se debe acceder a un bajo nivel de programación, por lo cual para dicho propósito, se deben usar operadores a nivel bit.
- **Optimización de memoria:** Si se está programando sobre algún dispositivo o microcontrolador con escasa memoria, se puede optimizar el uso de la misma empaquetando variables, esto quiere decir, en vez de definir 8 variables de tipo boolean, que ocuparían en total 64 bits, se puede utilizar solo 8 bit, tomando a cada bit como una variable booleana, en donde un 1 equivaldrá a verdadero, mientras que un 0 equivaldrá a falso.
- **Usos de máscaras de bits:** Las máscaras de bits son muy útiles porque permiten hacer directamente operaciones que, de otra manera, se necesitaría hacer un for y recorrer vectores.
- **Comunicación mediante TCP/IP:** Las direcciones IP no son más que bits, por lo cual, en situaciones que requieran trabajar a un bajo nivel, se deberán utilizar estos operadores.

- **Algoritmos de encriptación:** Los algoritmos de encriptación sirven para codificar o cifrar datos, y de esta manera protegerlos. Un ejemplo en donde se utiliza la encriptación es en contraseñas, al encriptar las mismas se las protege, para que ninguna persona pueda robarlas. Para realizar algoritmos de encriptación, generalmente se utilizan operadores a nivel bit.
- **Tratamiento de imágenes:** Las imágenes están conformadas por píxeles, los cuales pueden interpretarse como bits. En el tratamiento de imágenes, se realizan tareas de procesamiento arduas, por lo cual se requiere una gran eficiencia al trabajar con ellas. Para este tipo de aplicaciones se utilizan los operadores a nivel bit.
- **Elementos que usan banderas:** A menudo se trabajan con valores o elementos que tienen atributos llamados **Flags.** Estos atributos son bits, ubicados en algún lugar en particular de una cadena de bits, los cuales tienen un significado. Los archivos, por ejemplo, hacen la utilización de estas banderas, para indicar, entre otras cosas, si un archivo está oculto o no, está en modo de solo escritura, etc.
- **Arduino:** La plataforma Arduino, trabaja a nivel bit, por lo que hace uso de estos operadores.
- **Programación códec de audio y video:** Los códecs son mecanismos para transformar señales u ondas, de audio y video en bits. Para la programación de los mismos se deben utilizar operadores a nivel bit.
- **Algoritmos genéticos:** En la programación de algoritmos genéticos, se representa al ADN mediante bits. Para éste tipo de programación se requiere utilizar operadores a nivel bit.

Ejemplos de aplicación

Multiplicación por 2^n

Una forma de multiplicar un número por 2 a la enésima potencia equivale a hacer

num << potencia

Cociente de dividir por 2^n

Para dividir un número por 2 a la enésima potencia se debe hacer

num >> potencia

Resto de dividir por 2^n

Para obtener el resto de una división se puede hacer

(~((~0) << potencia)) & num

Paridad

Para saber si un número es par se puede hacer

num & 1

Si el resultado es 0, entonces el número será par y en caso contrario será impar.

Obtener la parte entera de un número en coma flotante

Para obtener la parte entera de un número en coma flotante se puede hacer

num | 0 o bien num ^ 0

Cifrado de mensajes

Se pueden cifrar mensajes utilizando la suma lógica XOR.

Nro = 231;
Para cifrar el número se puede hacer Nro ^= 21;

Para volver a obtener el número original se puede volver a hacer Nro ^= 21;

Pasar atributos a un método

Existen métodos, que aceptan solo un parámetro para definir su funcionamiento. Por ejemplo, una clase que tiene atributos correspondientes a TRABAJAR_LOCAL con el valor 10000000 que sirve para que la clase no se conecte a la red, y ENCRIPTACION_MD5 con el valor 000010000, que sirve para encriptar contraseñas, con las que la clase trabaje en MD5.

Si, como se mencionó, el método para definir el funcionamiento de la clase, acepta un solo parámetro, en primera instancia no se podría indicarle que trabaje como local y que a su vez encripte contraseñas en MD5.

Una forma de resolver esto, sería creando dos métodos en el que cada método active cada una de estas opciones individualmente, pero en el caso de que se tengan muchos más atributos, hacer esto resultaría ineficiente.
Otra opción sería crear un atributo llamado por ejemplo LOCAL_MD5, que representa la unión de las dos opciones posibles, pero se deberá crear una variable con todas las combinaciones posibles, que en este caso son solo 3 (tres) opciones posibles, pero en el caso de tener más atributos, serían muchísimas más.

Si en vez de eso se llama a la función de esta manera:

función(TRABAJAR_LOCAL | ENCRIPTACION_MD5);

Con un solo parámetro se le está indicando que trabaje de manera local y con encriptación MD5, y el valor de ese parámetro será 10001000. Entonces de esta manera, solo lo que habrá que hacer es comprobar éste parámetro para definir los modos que estén activados.

Este ejemplo de uso es **muy típico** y se suele ver mucho por ejemplo en la API gráfica Java 3D, y en otros motores gráficos.

Atributos de un archivo

Los valores de los atributos de ficheros o archivos, suelen tener valores correspondientes a ReadOnly, Hidden, System y Directory.

El valor ReadOnly sirve para saber si el fichero es de solo lectura, y tiene un valor de 1, el cual será el bit correspondiente al primer lugar.

El valor Hidden sirve para saber si un fichero está oculto o no, y tiene un valor de 2, por lo cual corresponde al bit número 2.

El valor System sirve para saber si un fichero, es exclusivamente del sistema, y tiene un valor de 4, por lo cual corresponde al bit número 3.

El valor Directory sirve para saber si un fichero es un directorio, y tiene un valor de 16, por lo cual corresponde al bit número 5.

Por lo tanto si un fichero, tiene el valor 10010, quiere decir que es un directorio y está oculto. El valor 10010, corresponde a 18 en sistema decimal.

Y si se quisiera consultar por algún valor en particular, por ejemplo si el fichero es visible o no, se puede verificar mediante la siguiente lógica:

atributos = 10010

Si (atributos & 00010 == 1) entonces
El fichero está oculto

Colores

En programación los colores se almacenan en 32 bits. Los 8 bits más significativos representan al canal Alpha o transparencia, los 8 siguientes a la cantidad del color rojo, luego los siguientes 8 bits representaran a la cantidad del color verde, y por último los 8 bits restantes representan a la cantidad del color azul.

Color = 01000000 00000101 00101011 11100011

Entonces, si se quisiera obtener cada canal por separado se debería hacer algo como lo siguiente:

Alpha = Color >> 24
Red = Color >> 16 & 0xFF
Green = Color >> 8 & 0xFF
Blue = Color & 0xFF

El valor 0xFF en hexadecimal corresponde al valor 11111111 en binario.

Operadores de asignación a nivel bit

De la misma manera que se pueden combinar los operadores aritméticos con los operadores de asignación, pueden combinarse los operadores de bit con éstos últimos, a excepción de la negación o complemento binario.

Operador	Descripción	Ejemplo	Equivalencia
|=	Suma lógica binaria combinada	a |= b	a = a | b
^=	Suma lógica exclusiva binaria combinada	a ^= b	a = a ^ b
&=	Producto lógico binario combinado	a &= b	a = a & b
<<=	Desplazamiento de bits hacia la izquierda combinado	a <<= b	a = a << b
>>=	Desplazamiento de bits hacia la derecha combinado	a >>= b	a = a >> b
>>>=	Desplazamiento de bits hacia la derecha sin signo combinado	a >>>= b	a = a >>> b

Operador de concatenación de cadenas

El operador de concatenación de cadenas "+", sirve para unir cadenas, o también para concatenar otro tipo de datos, como enteros con cadenas. El resultado siempre será una cadena.

Operador	Descripción	Ejemplo	Resultado
+	Concatenación	"Nombre: Juan;" + "Edad: " + 15	"Nombre: Juan; Edad: 15"

Operador Instanceof

Este operador sirve para saber si un objeto pertenece a una clase o no.

Operador	Descripción	Ejemplo	Resultado
instanceof	Indicador de instancia de una clase	auto instanceof Vehículo auto instanceof Mueble	true false

Tipos resultantes de operaciones mixtas

Cuando se realizan operaciones numéricas entre diferentes tipos, Java proporcionará un tipo resultante acorde al resultado, de modo que no provoque problemas de desbordamiento.

Si en la operación intervienen:

- Tipos byte, short, int, el resultado será de tipo int
- Tipo long, el resultado será de tipo long
- Tipo float, el resultado será de tipo float
- Tipo double, el resultado será de tipo double

Separadores

Es común que en el código de fuente de un programa se encuentren caracteres especiales como llaves, paréntesis, corchetes, entre otros. Estos caracteres se usan de separadores entre diferentes fragmentos de código, y sirven para definir la forma y la función del código.

Operador	Definición	Descripción
()	Paréntesis	Sirve para agrupar expresiones, alterar la precedencia normal de los operadores y su asociatividad, aislar expresiones condicionales, e indicar llamadas a funciones y señalar los parámetros de las mismas.
{}	Llaves	Sirve para contener los valores de matrices uni y multi dimensionales inicializados automáticamente, y definir el principio y el final de bloques de código, para clases, métodos y ámbitos locales.
[]	Corchetes	Sirve para declarar matrices uni y multidimensionales, y referenciar los valores de una matriz.

;	Punto y coma	Sirve para indicar el fin de una sentencia.
,	Coma	Sirve para separar identificadores de un mismo tipo en la declaración de variables, y separar parámetros en una función.
.	Punto	Sirve para separar nombres de paquetes, subpaquetes y clases, y acceder a los miembros de una clase.

Expresiones

Las expresiones son una combinación de operandos y operadores que generan un único resultado de un tipo determinado.

Precedencia de operadores

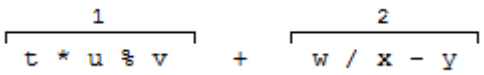
En una misma expresión, pueden encontrarse varios operadores, y el orden en que estos se evalúan puede determinar un resultado diferente según sea el caso.
El orden de precedencia de los operadores en Java es similar al orden de precedencia de los operadores en álgebra, ya que por ejemplo, si se toman las operaciones aritméticas, la multiplicación, división y resto se aplican primero, antes que la suma y resta. Se dice que estos tres operadores tienen el mismo nivel de precedencia, por lo cual, si se encuentran varios operadores de un mismo nivel de precedencia en una expresión, los operadores se evalúan de izquierda a derecha.

Ejemplo:

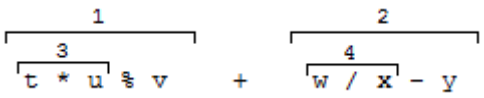
```
1   float z;
2   int t, u, v, w, x, y;
3
4   z = 0; t = 2; u = 5; v = 7;
5   w = 4; x = 2; y = 3;
6
7   z = t * u % v + w / x - y;
8   System.out.println(z);
9
10  z = t * (u % v + w) / (x - y);
11  System.out.println(z);
```

La línea 8 mostrará por consola 2.0, mientras que la línea 11 mostrará por consola -18.0.

Esto es justamente por la precedencia de los operadores.
En la línea 7 se tiene t * u % v + w / x – y, separando términos se tiene que:

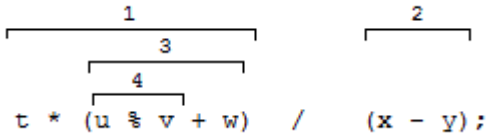


Si se analiza por separado cada término se tiene que:



Luego la suma de estos términos resultara igual a 2.

En la línea 10 se altera el orden de precedencia de los operadores utilizando paréntesis. Si se hace un análisis detallado del orden de precedencia se tiene que:



En donde hacer las operaciones en el orden indicado da como resultado -18.

A continuación se ilustra una tabla con todos los operadores vistos hasta el momento, en relación a su prioridad o precedencia en operaciones dentro de una expresión.

Prioridad	Operador	Tipo	Descripción
1	++ -- +, - ~ !	Aritmético Aritmético Aritmético A nivel bit Lógico	Incremento previo o posterior Incremento previo o posterior Suma unaria y resta unaria Cambio de bits Negación
2	*, /, %	Aritmético	Multiplicación, división y resto
3	+, - +	Aritmético Cadena	Suma y resta Concatenación de cadenas
4	<< >> >>>	A nivel bit A nivel bit A nivel bit	Desplazamiento de bits a la izquierda Desplazamiento de bits a la derecha con signo Desplazamiento de bits a la derecha sin signo
5	<, <= >, >= instanceof	Aritmético Aritmético Objeto	Menor que, menor o igual que Mayor que, mayor o igual que Comparación de tipos
6	== !=	Lógico Lógico	Igual Desigual
7	& &	A nivel bit Lógico	Cambio de bits AND Producto lógico
8	^ ^	A nivel bit Lógico	Cambio de bits XOR Suma exclusiva lógica
9	| |	A nivel bit Lógico	Cambio de bits OR Suma lógica
10	&&	Lógico	AND condicional
11	||	Lógico	OR condicional
12	= *=, /=, %= +=, -= <<=, >>= >>>= &=, ^=, |=	De asignación	Asignación Asignación con operación

Legibilidad y convenciones de escritura

Existen convenciones de escritura en programación, los cuales son adoptados por todo programador profesional y sirve, por ejemplo, para diferenciar clases de atributos. Es muy importante adoptar a este método de escritura ya que es un lenguaje común entre programadores y de esta manera se podrá tener una mejor comunicación con programadores a lo largo de todo el mundo y se podrá trabajar de una forma más limpia y prolija.

Clases: Las clases deben ser escritas en letras minúsculas, con la primera letra de cada palabra en mayúsculas, esta forma de escritura es llamada lomo de camello o CamelCase en inglés.

Ejemplos:

```
1   class MiClase;
2   class ClasePrincipal;
3   class Fisica;
4   class ManejadorDeGraficos;
```

Métodos: Los métodos deben ser verbos escritos en minúsculas con la primera letra de cada palabra en mayúsculas con excepción de la primera palabra del método.

Ejemplos:

```
1   dibujarRectangulo();
2   animar();
3   sumatoriaDeSaldos();
4   importarBaseDeDatos();
```

Atributos: Al igual que los métodos deben ser escritos en minúsculas con la primera letra de cada palabra en mayúsculas con excepción de la primera palabra del identificador del atributo.

Ejemplos:

```
1   double perimetro;
2   boolean datoBandera;
3   int saldosEnteros;
4   String baseDeDatos;
```

Constantes: Las constantes deben escribirse en su totalidad con mayúsculas y para separar palabras se debe usar el guión bajo (underscore) "_".

Ejemplos:

```
1   static final double PI = 3.141592;
2   static final int CONSTANTE_DE_EJEMPLO = 5;
```

```
3   static final float ACELERACION_DE_LA_GRAVEDAD = 9.81f;
4   static final float ENERGIA_EN_REPOSO_DEL_ELECTRON = .5110f;
```

Paquetes: Los nombres de los paquetes se escriben completamente en minúsculas.

Ejemplos:

```
1   package com.et35.miprimerprograma;
2   package com.box2d.fisica.*;
3   package com.reactiveacademy.librerias;
```

Estructuras de control: Cuando una sentencia forma parte de una estructura de control como por ejemplo de flujo como for o while, es necesario escribir los corchetes ({}), aunque solo se tenga una línea de código:

```
1   for (int i = 0; i<TOPE; i++){
2       Expresión;
3       Expresión;
4   }
```

Espacios: Los espacios son necesarios para entender el código de mejor manera, para mantenerlo limpio, ordenado y distinguir el principio y el final de estructuras de control. Cada vez que se escribe algo dentro de una llave "{", se hace luego de una tabulación:

```
1   if (bandera) {
2     bandera = false;
3     if (n1>n2) {
4       Systen.out.println("El nro " + n1 + " es mayor al nro " + n2");
5     } else if (n2>n1) {
6       Systen.out.println("El nro " + n2 + " es mayor al nro " + n1");
7     } else {
8       Systen.out.println("Los dos números son iguales");
9     }
10  }
```

Comentarios: Los comentarios son útiles y recomendados para explicar ciertas partes del código o hacer aclaraciones sobre el mismo. Especialmente si es un proyecto en el que pueda trabajar más de un programador.

```
1   /* Autor: Pablo Ezequiel Jasinski
2      Fecha de creación: 01/02/2015 */
3
4   // Función para pasar ángulos de grados a radianes
5   private double degToRad(double deg){
6     return ((deg*Math.PI)/180);
7   }
```

Los comentarios son líneas de código ignoradas por el compilador y existen tres formas de hacer comentarios.

La primera forma es con // lo que se conoce como **comentario de fin de línea**. Este tipo de comentarios son útiles para escribir solo una línea, si se desea escribir más de una línea se debe escribir // por cada línea que se quiera comentar.

La segunda forma sirve para comentar varias líneas y se delimitan mediantes los caracteres /* para empezar un comentario y */ para finalizarlo.

Existe una tercera forma de comentar código, que por el momento no se va a detallar demasiado, que se conoce como comentarios **Javadoc** donde los comentarios están delimitados por /** y */, y básicamente mediante estos comentarios se generan archivos HTML con la documentación del programa realizado. Hoy en día es el formato preferido por la industria.