Unidad 1

DIPLOMATURA EN PROGRAMACION JAVA

Capítulo 2

Visibilidad de Variables y Expresiones

Visibilidad de Variables y Expresiones

En Este Módulo

- Variables y Visibilidad
- La palabra clave this
- Los Métodos Estáticos y su Visibilidad
- Declaración de Constantes
- Operadores
- Promoción de Expresiones en las Conversiones de Tipo
- Concatenación de Strings
- Clases de Envoltorio
- Conversión automática de tipos primitivos en objetos: autoboxing
- Convirtiendo Strings a valores numéricos

La palabra clave this

Esta palabra clave tiene como finalidad almacenar la referencia del objeto que se encuentra en ese momento en ejecución y se utiliza para encontrar los elementos que pertenecen a éste. Como se mencionó anteriormente, este valor se almacena en el stack para cada método en ejecución y es por esta referencia que se encuentran las variables de instancia de un objeto.

Sólo los elementos de un objeto, métodos y atributos, poseen asociada una referencia del tipo **this**, por lo tanto se lo puede utilizar para resolver ambigüedades.

Normalmente, dentro del cuerpo de un método de un objeto se puede referir directamente a las variables miembros del objeto. Sin embargo, algunas veces es necesario evitar algún tipo de ambigüedad acerca del nombre de un atributo y uno o más de los argumentos del método que tengan el mismo nombre.

Por ejemplo, si una clase tiene declaradas variables de instancia que tienen el mismo nombre que los argumentos de un método, utilizando **this** se resuelve la ambigüedad porque los métodos poseen parámetros y variables locales que se alojan en el stack, por lo tanto nunca podrán tener estos elementos un **this** asociado.

Ejemplo

```
public class Ejemplo{
	Ejemplo(int a){ atrib1 = a;}
	Ejemplo(char b){ atrib2 = b;}
	Ejemplo (char atrib2,int atrib1){
		this.atrib1 = atrib1;
		this.atrib2 = atrib2;
	}

	private int atrib1;
	private char atrib2;
	public void metodo(){
		//[sentencias;]
		atrib2 = 'a';
	}
}
```

En el ejemplo anterior se muestra la resolución de visibilidad dentro del constructor de la clase. Si no se utilizará la palabra clave **this**, y como respecto de la visibilidad siempre tiene prioridad los elementos de visibilidad local, no se podrían diferenciar los parámetros (locales) de los atributos del objeto.

Variables y visibilidad

Las variables locales son:

- Definidas dentro de un método (se las llama locales, automáticas, temporarias o de stack)
- Se crean cuando se ejecuta un método y se destruyen cuando este termina
- Pueden almacenar valores o referencias

- Las variables que almacenan referencias siempre se deben inicializar con un operador **new** antes de utilizarse

La duda que se presenta siempre a todo programador es como "saben" dos objetos del mismo tipo cuando se están ejecutando cuáles son sus variables de instancia y cuáles no le corresponden. En realidad, lo que sucede por detrás, es que la máquina virtual asigna a todo método que fue definido en una clase un primer parámetro oculto (**this**) que tiene almacenada la referencia del objeto que se esta ejecutando actualmente. ¿Cómo puede saber eso? La respuesta es simple: la variable de referencia que se utiliza con la notación de punto almacena dicha referencia a memoria, por lo tanto, lo único que hace es incluirla como primer parámetro por defecto de todo método a ejecutarse de un objeto.

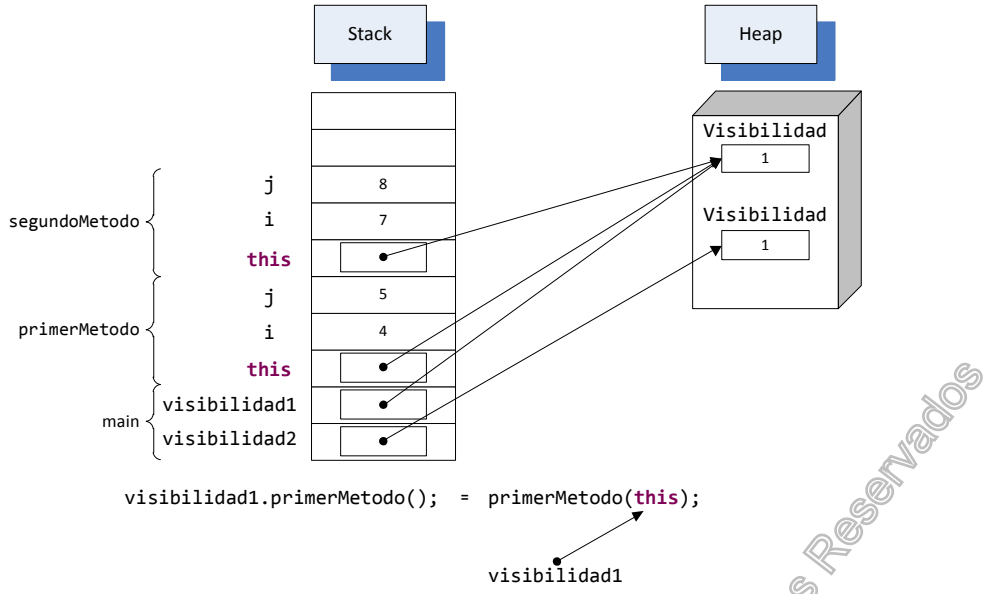
Ejemplo

```
package visibilidad;

public class Visibilidad {
	private int i = 1;

	public void primerMetodo() {
		int i = 4, j = 5;
		this.i = i + j;
		segundoMetodo(7);
	}

	public void segundoMetodo(int i) {
		int j = 8;
		this.i = i + j;
	}

	public static void main(String[] args) {
		Visibilidad visibilidad1 = new Visibilidad();
		Visibilidad visibilidad2 = new Visibilidad();
		visibilidad1.primerMetodo();
		visibilidad2.primerMetodo();
	}
}
```

Los siguientes gráficos muestran la evolución del stack y el lugar al que se apunta en el heap para cada secuencia de llamados. Por ejemplo

```
		visibilidad1.primerMetodo();
```

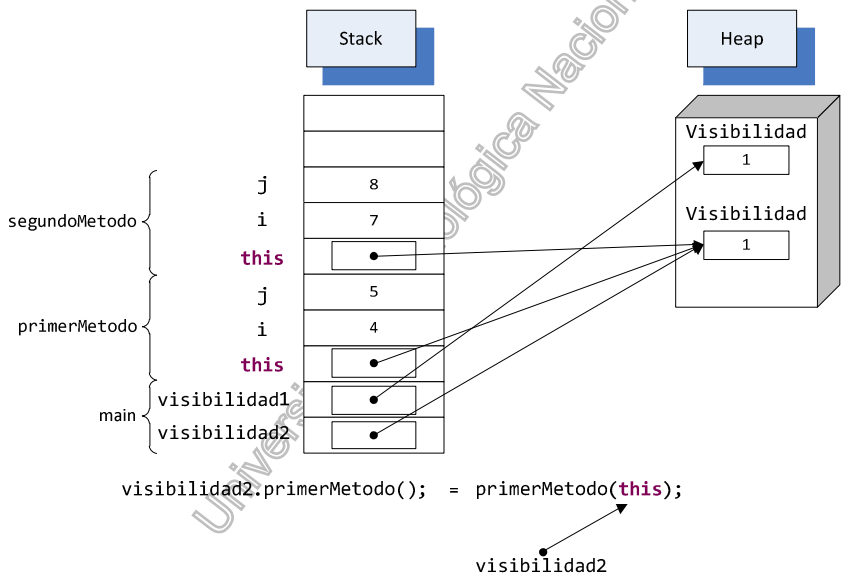
Genera



Mientras que

```
visibilidad2.primerMetodo();
```

Genera



Los métodos estáticos y su visibilidad

Este tipo de métodos se diferencian de cualquier otro tipo declarado en una clase. Como se demostró anteriormente los métodos pasan encubiertamente una referencia **this** como primer parámetro en cualquier método perteneciente a un objeto y la razón es que éste encuentre las variables de instancia que pertenecen a "este" objeto para poder operar con ellas. Esto es cierto salvo para los métodos estáticos

> *Los métodos estáticos se diferencian del resto porque no tienen una referencia* **`this`** *asociada*

La principal consecuencia es que no pueden acceder a ningún elemento de un objeto en tiempo de ejecución porque al no tener forma de encontrarlos mediante una referencia, "no sabe que existen".

Es claro que si se le pasa como parámetro una referencia **`this`** a un método estático, este podrá acceder a los elementos del objeto limitado siempre por la visibilidad que los mismos posean.

Respecto justamente de la visibilidad, ¿cómo es afectada la visibilidad en un método estático? Como se mencionó en el módulo anterior, las visibilidades están determinadas por los bloques en los que se realizan las declaraciones. Al estar los métodos estáticos definidos dentro de clases, tienen visibilidad dentro de las mismas y para poder accederlos hay que resolver ese alcance.

La forma de resolver visibilidad para acceder un método estático es utilizar el nombre de la clase seguido de la notación de punto. Esto sumado al hecho que no tiene un **`this`** asociado permite realizar la siguiente afirmación

> *Todo método estático puede ser accedido mediante el nombre de la clase que lo contiene, seguido de la notación de punto y el nombre del mismo, sin la necesidad de crear un objeto para invocarlo.*

Este hecho tiene importancia a nivel de diseño de clases como se verá posteriormente. Sin embargo ya se ha visto una utilidad de esta característica, el método **main**. Java no puede iniciar un programa declarando un objeto de una clase para acceder al método **main** que esta posea. Al ser éste estático, no tiene necesidad de crear ningún objeto, sólo saber la clase en la que se encuentra para resolver la visibilidad.

Declaración de constantes

Para crear un atributo miembro constante en Java se debe utilizar la palabra clave final en la declaración. La siguiente declaración define una constante llamada AVOGADRO cuyo valor es el número de Avogadro (6.023×10^{23}) y no puede ser cambiado:

```
class Avo {
    final double AVOGADRO = 6.023e23;
}
```

Por convención, los nombres de los valores constantes se escriben completamente en mayúsculas. Si un programa intenta cambiar una constante, el compilador muestra un mensaje de error similar al siguiente, y no compila el programa.

```
The final field Avo.AVOGADRO cannot be assigned
```

Operadores

Los operadores realizan operaciones entre uno, dos o tres operandos.

Los operadores que requieren un operador se llaman operadores unarios. Por ejemplo, **++** es un operador *unario* que incrementa el valor su operando en uno.

Los operadores que requieren dos operandos se llaman operadores *binarios*. El operador **=** es un operador binario que asigna un valor del operando derecho al operando izquierdo.

Existe un único operador ternario que trabaja con tres operandos y funciona como un condicional y dependiendo de cómo evalúa la condición *devuelve un valor u otro*. Este operador es **:?**

Los operadores unarios en Java pueden utilizar la notación de prefijo o de sufijo (también conocidas como pre y post, por ejemplo, un pre incremento de la variable i es ++i, mientras que un post incremento es i++). La notación de prefijo significa que el operador aparece antes de su operando, por ejemplo,

operador operando

La notación de sufijo significa que el operador aparece después de su operando:

operando operador

Todos los operadores binarios de Java tienen la misma notación, es decir aparecen entre los dos operandos:

op1 operator op2

Además de realizar una operación también devuelve un valor. El valor y su tipo dependen del tipo del operador y del tipo de sus operandos. Por ejemplo, los operadores aritméticos (realizan las operaciones de aritmética básica como la suma o la resta) devuelven números, el resultado típico de las operaciones aritméticas.

El tipo de datos devuelto por los operadores aritméticos depende del tipo de sus operandos: si se suma dos enteros, se obtiene un entero. Se dice que una operación evalúa su resultado.

Es muy útil dividir los operadores Java en las siguientes categorías: aritméticos, relacionales y condicionales, lógicos y de desplazamiento y de asignación.

Operadores aritméticos

El lenguaje Java soporta varios operadores aritméticos, incluyendo

Operador	Descripción
+	suma
-	resta
*	multiplicación
/	división

%	módulo

En todos los números enteros y de coma flotante. Por ejemplo, se puede utilizar este código Java para sumar dos números:

```
sumarEsto + aEsto
```

O este código para calcular el resto de una división:

```
dividirEsto % porEsto
```

Esta tabla resume todas las operaciones aritméticas binarias en Java:

Operador	Uso	Descripción
+	op1 + op2	Suma op1 y op2
-	p1 - op2	Resta op2 de op1
*	op1 * op2	Multiplica op1 y op2
/	op1 / op2	Divide op1 por op2
%	op1 % op2	Obtiene el resto de dividir op1 por op2

Nota: El lenguaje Java extiende la definición del operador + para incluir la concatenación de cadenas (Strings).

Los operadores + y - tienen versiones unarias que seleccionan el signo del operando:

Operador	Uso	Descripción
+	+op	Indica un valor positivo
-	- op	Niega el operando

Además, existen dos operadores de incremento y decremento aritméticos, ++ que incrementa en uno su operando, y -- que decrementa en uno el valor de su operando.

Operador	Uso	Descripción
++	op ++	Incrementa op en 1; evalúa el valor antes de incrementar
++	++ op	Incrementa op en 1; evalúa el valor después de incrementar
--	op --	Decrementa op en 1; evalúa el valor antes de decrementar
--	-- op	Decrementa op en 1; evalúa el valor después de decrementar

Operadores de asignación

Se puede utilizar el operador de asignación **=**, para asignar un valor a otro.

Además del operador de asignación básico, Java proporciona varios operadores de asignación que permiten realizar operaciones aritméticas, lógicas o de bits y una operación de asignación al mismo tiempo.

Específicamente, suponiendo que se quiere añadir un número a una variable y asignar el resultado dentro de ella misma, se puede hacer

```
i = i + 2;
```

Se puede lograr el mismo resultado en la sentencia utilizando el operador **+=**.

```
i += 2;
```

Las dos líneas de código anteriores son equivalentes.

La siguiente tabla lista los operadores de asignación y sus equivalentes:

Operador	Uso	Equivale a
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	p1 *= op2	op1 = op1 * op2
/=	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
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Operadores relacionales

Los valores relacionales comparan dos valores y determinan la relación entre ellos. Por ejemplo, **!=** devuelve `true` si los dos operandos son distintos.

La siguiente tabla resume los operadores relacionales de Java:

Operador	Uso	Devuelve true si
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 son distintos

Operadores lógicos o condicionales

Se utilizan para construir expresiones de decisión complejas. Uno de estos operadores es **&&** que realiza la operación **Y** lógico o booleano. Por ejemplo, se puede utilizar dos operadores relacionales diferentes y evaluar la totalidad de la expresión uniéndola con **&&** para determinar si ambas relaciones son ciertas. La siguiente línea de código utiliza esta técnica para determinar si un

valor esta entre dos límites, esto es, para determinar si el índice es mayor que 0 o menor que NUM_ENTRADAS (que se ha definido previamente como un valor constante):

```
0 < index && index < NUM_ENTRADAS
```

Se puede observar que en algunas situaciones, el segundo operando de un operador relacional no será evaluado. Consideremos esta sentencia:

```
((contador > NUM_ ENTRADAS) && (System.in.read() != -1))
```

Si **contador** es menor que **NUM_ ENTRADAS** en la parte izquierda del operando **&&**, la parte derecha no se evalúa. El operador **&&** sólo devuelve `true` si los dos operandos son verdaderos. Por eso, en esta situación se puede determinar el valor de **&&** sin evaluar el operador de la derecha y en un caso como este, Java ignora el operando de la derecha. Así no se llamará a System.in.read() y no se leerá un carácter de la entrada estándar.

Los operadores condicionales son:

Operador	Uso	Devuelve true si
&&	op1 && op2	op1 y op2 son verdaderos
||	op1 || op2	uno de los dos es verdadero
!	! op	op es falso

El operador **&** se puede utilizar como un sinónimo de && si ambos operadores son bits. Similarmente, | es un sinónimo de || si ambos operandos son bits. En definitiva, estos operadores están diseñados para actuar como sus contrapartidas, pero en lugar de evaluar todo el operando, lo hace bit a bit entre cada uno de los operandos. Para entender mejor lo afirmado, se realiza la siguiente clasificación

Operadores lógicos a nivel de bit entre operandos

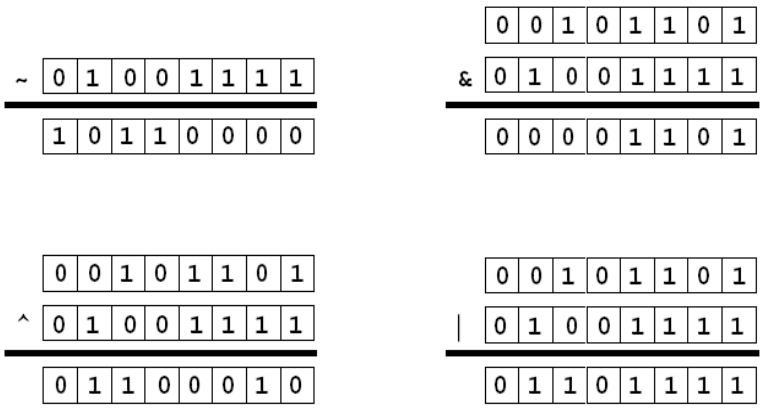
Operador	Descripción
~	NOT
|	OR
&	AND
^	XOR

Operadores lógicos a nivel de datos

Operador	Descripción
&&	AND
||	OR
!	NOT

La siguiente tabla resume una explicación de los operadores a nivel de bit que realizan funciones lógicas para cada uno de los pares de bits de cada operando

Operador	Operación	Explicación
&	op1 & op2	operador a nivel de bit **Y**
|	op1 | op2	operador a nivel de bit **O**
^	op1 ^ op2	operador a nivel de bit **O excluyente**
~	~ op	operador a nivel de bit **complemento**



Ejemplo

La función "**y**" (**and**) activa el bit resultante si los dos operandos son 1.

op1	op2	Resultado
0	0	0
0	1	0
1	0	0
1	1	1

Suponiendo que se quiere evaluar los valores 12 **and** 13:

13 & 12 = 12

El resultado de esta operación es 12. La causa es que la representación binaria de 12 es 1100 y la de 13 es 1101. La función **and** activa los bits resultantes cuando los bits de los dos operandos son 1, de otra forma el resultado es 0. Entonces si se coloca en línea los dos operandos y se realiza la función **Y**, se puede ver que los dos bits de mayor peso (los dos bits situados más a la izquierda de cada número) son 1 y así el bit resultante de cada uno es 1. Los dos bits de menor peso se evalúan a 0 porque al menos uno de los dos operandos es 0:

	Binario	**Decimal**
	1 1 0 1	13
&	1 1 0 0	12
	1 1 0 0	12

Ejemplo

```
package operadores;
public class OperadorBitY {
	public static void main(String[] args) {
		int a = 12, b = 13, c = 0;
		c = b & a;
		System.out.println("c: " + c);
	}
}
```

La salida es

```
		c: 12
```

El operador | realiza la operación **O inclusivo** y el operador ^ realiza la operación **O exclusivo**. **O inclusivo** significa que si uno de los dos operandos es 1 el resultado es 1:

op1	op2	Resultado
0	0	0
0	1	1
1	0	1
1	1	1

O exclusivo significa que si los dos operandos son diferentes el resultado es 1, de otra forma el resultado es 0:

op1	op2	Resultado
0	0	0
0	1	1
1	0	1
1	1	0

Y finalmente el operador **complemento** invierte el valor de cada uno de los bits del operando: si el bit del operando es 1 el resultado es 0 y si el bit del operando es 0 el resultado es 1.

op	Resultado
1	0
0	1

Operadores de desplazamiento

Los operadores de desplazamiento permiten realizar manipulación de los bits en los datos. Esta tabla resume los operadores lógicos y de desplazamiento disponibles en el lenguaje Java:

Operador	Uso	Descripción
>>	op1 >> op2	desplaza a la derecha op2 bits de op1
<<	op1 << op2	desplaza a la izquierda op2 bits de op1
>>>	op1 >>> op2	desplaza a la derecha op2 bits de op1(sin signo)

Los tres operadores de desplazamiento simplemente desplazan los bits del operando de la izquierda el número de posiciones indicadas por el operador de la derecha. Los desplazamientos ocurren en la dirección indicada por el propio operador.

Ejemplo

```
13 >> 1;
```

Desplaza los bits del entero 13 una posición a la derecha. La representación binaria del número 13 es 1101. El resultado de la operación de desplazamiento es 110 o el número 6 en base decimal. Se puede observar que el bit situado más a la derecha (el de menor peso) desaparece. Un desplazamiento a la derecha de un bit es equivalente, pero más eficiente que, dividir el operando de la izquierda por dos y desestimar el resto. Un desplazamiento a la izquierda es equivalente a multiplicar por dos.

Ejemplo

Desplazamiento a derecha:

Operación	Equivale	Resultado
128 >> 1	$128/2^1$	64
256 >> 4	$256/2^4$	16
-256 >> 4	$-256/2^4$	-16

Desplazamiento a izquierda:

Operación	Equivale	Resultado
128 << 1	$128 * 2^1$	256
16 << 2	$16 * 2^2$	64
-16 << 2	$16 * 2^2$	-64

Ejemplo

Operación	Resultado Binario																														Resultado Decimal
2423	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	1	0	1	1	1	2423
-2423	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	0	0	0	1	0	0	1	-2423

Operación	Resultado Binario	Resultado Decimal
2423 >> 6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1	37
-2423 >> 6	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0	-38
2423 >>> 6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1	37
-2423 >>> 6	0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0	67108826
2423 << 6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 1 0 1 1 1 0 0 0 0 0 0	155072
-2423 << 6	1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0	-155072

Prioridad de los operadores

La siguiente figura ilustra los operadores de Java y su precedencia

Prioridad

Asociación	Operadores
Separador	. [] () ; ,
D a I	++ -- + - ~ ! (Tipo de Datos)
I a D	* / %
I a D	+ -
I a D	<< >> >>>
I a D	< > <= >= instanceof
I a D	== !=
I a D	&
I a D	^
I a D	|
I a D	&&
I a D	||
D a I	<expresión booleana>? <expresión si es verdadero>:<expresión si es falso>
D a I	= *= /= %= += -= <<= >>= >>>= &= ^= |=
I a D	++ -- (Post incremento y decremento)

Nota: el operador unario de incremento y decremento cambio su prioridad dependiendo del lugar en el que se encuentre. Cuando se encuentra previo al operando al que afecta es uno de los operadores de mayor prioridad. Sin embargo cuando esta post operando pasa a ser el de menor prioridad, *incluyendo la asignación*.

El siguiente ejemplo muestra el comportamiento de los operadores unarios según su prioridad para realizar operaciones y asignaciones.

Ejemplo

```
package operadores;
public class OperadoresUnarios {
	public static void main(String[] args) {
		int a = 4, b = 5, c = 6, d = 0;
		d = ++a - b-- - c;
		System.out.println("Valor de d: " + d);
		System.out.println("Valor de a: " + a);
		System.out.println("Valor de b: " + b);
		System.out.println("-----------");
		d = a++ - --b - c;
		System.out.println("Valor de d: " + d);
		System.out.println("Valor de a: " + a);
		System.out.println("Valor de b: " + b);
	}
}
```

Notar como se realizan primero las operaciones de pre incremento o decremento para luego resolver el resto de la expresión por medio de la asociatividad explicada. La salida del programa es:

```
Valor de d: -6
Valor de a: 5
Valor de b: 4
-----------
Valor de d: -4
Valor de a: 6
Valor de b: 3
```

El siguiente ejemplo destaca también las prioridades en las asignaciones para los operadores binarios.

Ejemplo

```
package operadores;
public class OperadoresBinarios {
	public static void main(String[] args) {
		int a = 4, b = 5, c = 6, d = 0;
		d = a + b * c;
		System.out.println("Valor de d: " + d);
		System.out.println("-----------");
		d += a;
		System.out.println("Valor de d: " + d);
		System.out.println("-----------");
		d -= b * c;
		System.out.println("Valor de d: " + d);
		System.out.println("-----------");
		d %= c;
		System.out.println("Valor de d: " + d);
		System.out.println("-----------");
		d <<= c;
		System.out.println("Valor de d: " + d);
		System.out.println("Valor de c: " + c);
		System.out.println("-----------");
```

```
            d >>= c++;
            System.out.println("Valor de d: " + d);
            System.out.println("Valor de c: " + c);
      }
}
```

La salida del programa es

```
Valor de d: 34
-----------
Valor de d: 38
-----------
Valor de d: 8
-----------
Valor de d: 2
-----------
Valor de d: 128
Valor de c: 6
-----------
Valor de d: 2
Valor de c: 7
```

Por último se muestra el uso del operador ternario. Este operador debe pensarse como una sentencia condicional en dónde si lo que se afirma en el primer operando es cierto, se retorna el segundo valor, sino, el tercero

Ejemplo

```
package operadores;
public class OperadorTernario {
      public static void main(String[] args) {
            int a = 4, b = 5, c = 6, d = 0;
            d = a > b ? c : b;
            System.out.println("Valor de d: " + d);
            d = a < b ? c : b;
            System.out.println("Valor de d: " + d);
            d = a < b ? c : ++b;
            System.out.println("Valor de b: " + b);
            System.out.println("Valor de d: " + d);
            d = a < b ? c : b--;
            System.out.println("Valor de d: " + d);
            System.out.println("Valor de b: " + b);
      }
}
```

La salida del programa es:

```
Valor de d: 5
Valor de d: 6
Valor de b: 5
Valor de d: 6
Valor de d: 6
Valor de b: 5
```

Promoción de expresiones en las conversiones de tipo

Las variables se promueven automáticamente a tipos más grandes (como por ejemplo, un **int** a **long**). Esto se debe a que Java permite que si una variable se asigna a otra del mismo tipo (por ejemplo, dos tipos enteros) automáticamente verifica que el tamaño en donde se va a asignar el valor sea mayor o igual que el mismo.

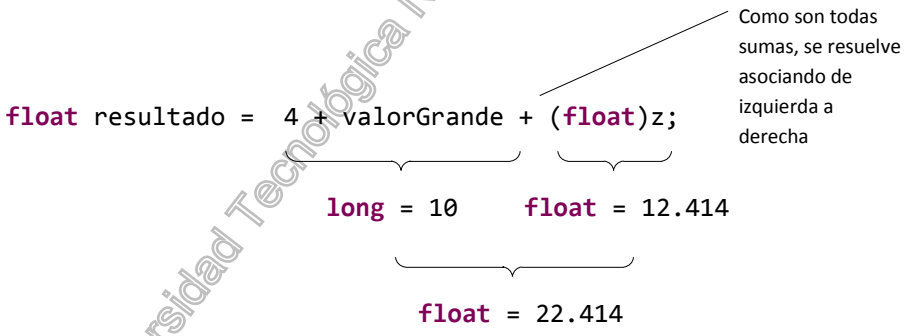
Por lo tanto, las asignaciones de expresiones son compatibles si el tipo de variable que recibe el valor es al menos del mismo tamaño en bits que el resultado que la expresión arroja

```
long valorGrande = 6; // 6 es un tipo entero, esta bien
int valorPequeño = 99L; // 99L es un long, esta mal
double z = 12.414F; // 12.414F es float, esta bien
float z1 = 12.414; // 12.414 es double, esta mal
```

Las promociones ocurren también en expresiones aritméticas como pasos intermedios. Por ejemplo, si se tiene el siguiente código

```
long valorGrande = 6; // 6 es un tipo entero, esta bien
double z = 12.414F; // 12.414F es float, esta bien
float resultado =  4 + valorGrande + (float)z;
```

La forma en que se realizan los pasos intermedios para resolver la expresión se muestran en el siguiente gráfico



Concatenación de Strings

Java permite concatenar cadenas fácilmente utilizando el operador **+**. Este operador puede crear un nuevo String para resolver una concatenación. Ejemplo

```
String titulo = "Dr.";
String nombre = "Pedro" + " " + "Ramirez";
String saludo = titulo + " " + nombre;
```

En una concatenación, si algún elemento no es String, se lo convierte automáticamente, pero se debe tener en cuenta que al menos un elemento debe ser de tipo String. El lenguaje trata a este

tipo en particular como uno superior respecto de los demás y realiza una *promoción* de otro tipo de datos a String

El siguiente fragmento de código concatena tres cadenas

```
"La entrada tiene " + contador + " caracteres."
```

Dos de las cadenas concatenadas son literales: "La entrada tiene " y " caracteres.". El tercer String, el del medio, es realmente un entero que primero se convierte a tipo String y luego se concatena con las otras cadenas.

Clases de Envoltorio

Las clases de envoltorio sirven para utilizar valores almacenados en datos primitivos como si fueran objetos. La razón de estos es que los datos primitivos no son manejados como objetos internamente, por lo tanto para utilizarlos así, se deben crear objetos que en su construcción reciben como argumento el dato primitivo y permiten el manejo del valor como si fueran objetos.

Las variables primitivas tienen mecanismos de reserva y liberación de memoria más eficaces y rápidos que los objetos por lo que deben usarse datos primitivos en lugar de sus correspondientes envolturas siempre que se pueda

Java provee una clase de envoltorio para cada tipo de dato primitivo, como se muestra en la siguiente tabla

Tipo de dato primitivo	Clase de envoltorio
`boolean`	Boolean
`byte`	Byte
`char`	Character
`short`	Short
`int`	Integer
`long`	Long
`float`	Float
`double`	Double

Las clases de envoltorio son útiles también para las conversiones de tipos porque definen métodos para este fin. Cada tipo tiene una serie de métodos específicos para la conversión que se necesite realizar

A modo de ejemplo se presentan algunos de los métodos de la clase Integer, pero existen métodos similares en todas las clases de envoltorio para los diferentes tipos.

Método	Descripción
Integer(`int` valor)	Constructor a partir de un `int`
Integer(String valor)	Constructor a partir de un String
`int` intValue() / `byte` byteValue() / `float` floatValue() . . .	Devuelve el valor en distintos formatos, `int`, `long`, `float`, etc.

boolean equals(Object obj)	Devuelve true si el objeto con el que se compara es un Integer y su valor es el mismo.
static Integer getInteger(String s)	Devuelve un Integer a partir de una cadena de caracteres.
static int parseInt(String s)	Devuelve un int a partir de un String.
static String toBinaryString(**int** i)	Convierte un entero a su representación en binario con formato de un String.
static String toOctalString(**int** i)	Convierte un entero en su representación en octal con formato de un String
static String toHexString(**int** i)	Convierte un entero en su representación en hexadecimal con formato de un String
static String toString(**int** i)	Convierte un entero en su representación con formato de un String
String toString()	Convierte un entero en su representación con formato de un String
static Integer valueOf(String s)	Devuelve un Integer a partir de un String.

Conversión automática de tipos primitivos en objetos: autoboxing

Si se necesita cambiar los tipos de datos primitivos en su objeto equivalente (operación denominada **boxing**), hay que utilizar las clases envoltorio.

Igualmente, para obtener el tipo de dato primitivo a partir de la referencia del objeto (lo que se denomina **unboxing**), también necesita usar los métodos de las clases envoltorio. Todas estas operaciones de conversión pueden complicar el código y dificultar su comprensión. En la versión 5.0 de J2SE se ha introducido una función de conversión automática denominada **autoboxing** que permite asignar y recuperar los tipos primitivos sin necesidad de usar clases envoltorio.

El ejemplo siguiente contiene dos casos sencillos de conversión y recuperación automática de primitivos (autoboxing y autounboxing).

Ejemplo

```
package boxing;
public class Autoboxing {
	public static void main(String[] args) {
		int entero = 420;
		Integer envoltorio = entero; // esto se denomina autoboxing
		System.out.println(envoltorio);
		int p2 = envoltorio; // esto se denomina autounboxing
		System.out.println(p2);
	}
}
```

Un compilador de J2SE versión 5.0 o superior ahora creará el objeto de envoltorio automáticamente cuando se asigne un primitivo a una variable del tipo de la clase de envoltorio.

Asimismo, el compilador extraerá el valor primitivo cuando realice la asignación de un objeto de envoltorio a una variable de tipo primitivo.

Esto puede hacerse al pasar parámetros a métodos o, incluso, dentro de las expresiones aritméticas.

Nota: No abusar de la función de autoboxing. El rendimiento se ve afectado cada vez que se utiliza la conversión o recuperación automática de tipos primitivos. La combinación de tipos primitivos y objetos de envoltorio en expresiones aritméticas dentro de un ciclo cerrado podría tener efectos negativos sobre el rendimiento y la velocidad de transmisión de datos de las aplicaciones.

Convirtiendo Strings a valores numéricos

Una necesidad común de un desarrollador Java es convertir cadenas a valores numéricos para luego manipularlos como un dato primitivo (por ejemplo, un **int**, **float**, **doublé**, etc…). Existen numerosas situaciones en la que esta necesidad se presenta (lecturas de archivos, parámetros de línea de comando, etc…), o tan sólo tomar un valor ingresado en el campo de texto de una interfaz gráfica para usuarios.

Para convertir un String a un dato primitivo, se debe comprender el uso de las clases de envoltorio. Cada tipo primitivo tiene asociado una clase de este tipo, como se muestra en la tabla anterior. Estas clases permiten tratar a los datos primitivos como objetos, dando además el beneficio de servicios extras para que se puedan manipular estos objetos a través de sus métodos en la forma apropiada para cada tipo primitivo.

Todas las conversiones excepto las del tipo **boolean**, se realizan con métodos que tienen nombres similares aunque diferentes,

Tipo Primitivo	Método de Conversión
byte	`Byte.parseByte(unString)`
short	`Short.parseShort(unString)`
int	`Integer.parseInt(unString)`
long	`Long.parseLong(unString)`
float	`Float.parseFloat(unString)`
double	`Double.parseDouble(unString)`
boolean	`Boolean.valueOf(unString).booleanValue()`

Cada una de estas clases de envoltorio, excepto la clase Character, tiene un método que permite convertir un tipo primitivo a String (o sea, la operación inversa). Lo único que se debe hacer es llamar al correspondiente método de la clase de envoltorio apropiada para convertir el tipo de dato.

Ejemplo

```
String miString = "12345";
int miInt = Integer.parseInt(miString);
```

Estas sentencias convierten el contenido de una variable de tipo String llamada **miString** a un tipo primitivo **int** llamado **miInt**. Si bien la conversión es simple, se debe tener en cuenta que para cada tipo de datos esta involucrado un único nombre de método en cada clase de envoltorio.

Existe una excepción, la clase Character no tiene un método de este tipo, en su lugar, se debe especificar que caracter se quiere rescatar del String indicando la posición que este ocupa en él con el método **charAt**. Si por ejemplo, se quiere obtener el caracter "o" en una cadena "Hola", la forma es la siguiente:

Ejemplo

```
String hello = "Hola";
char e = hello.charAt(1);
```

Si en cualquiera de los casos mencionados no se puede convertir el tipo String, se produce un error en tiempo de ejecución.

El siguiente es un ejemplo integrador de los operadores de desplazamiento a nivel de bits y el método estático de la clase Integer para mostrar números binarios en formato String

Ejemplo

```
package operadores;

public class OperadoresDesplazamiento {
	public static void main(String[] args) {
		int d = 2423;
		int resultado = d;

		System.out.println("Decimal:" + resultado + "\t\tResultado: " +
			Integer.toBinaryString(resultado));
		resultado = -d;
		System.out.println("Decimal:" + resultado + "\t\tResultado: " +
			Integer.toBinaryString(resultado));
		resultado = d >> 6;
		System.out.println("Decimal:" + resultado + "\t\tResultado: " +
			Integer.toBinaryString(resultado));
		resultado = -d >> 6;
		System.out.println("Decimal:" + resultado + "\t\tResultado: " +
			Integer.toBinaryString(resultado));
		resultado = d >>> 6;
		System.out.println("Decimal:" + resultado + "\t\tResultado: " +
			Integer.toBinaryString(resultado));
		resultado = -d >>> 6;
		System.out.println("Decimal:" + resultado + "\tResultado: " +
			Integer.toBinaryString(resultado));
		resultado = d << 6;
		System.out.println("Decimal:" + resultado + "\t\tResultado: " +
			Integer.toBinaryString(resultado));
		resultado = -d << 6;
```

```
            System.out.println("Decimal:" + resultado + "\t\tResultado: " +
                  Integer.toBinaryString(resultado));
      }
}
```

La salida obtenida es:

```
Decimal:2423        Resultado: 100101110111
Decimal:-2423       Resultado: 11111111111111111111011010001001
Decimal:37          Resultado: 100101
Decimal:-38         Resultado: 11111111111111111111111111011010
Decimal:37          Resultado: 100101
Decimal:67108826    Resultado: 11111111111111111111011010
Decimal:155072      Resultado: 100101110111000000
Decimal:-155072     Resultado: 11111111111111011010001001000000
```

Nota: Las clases de envoltorio y String generan objetos inmutables, esto quiere decir que ***si se le cambia el valor que contienen generan un nuevo objeto*** descartando el anterior para ser recolectado como basura del heap.