Unidad Didáctica VIII

Introducción

*Después de su iniciación a Java, **Víctor** está en condiciones de intentar hacer su primera aplicación con ciertas garantías. **José** ha decidido que lo mejor será que colabore con **Carmen** en la actualización de una aplicación que acaba de llegar y tienen que terminar cuanto antes por necesidades del cliente. Para ello **Víctor** tiene que demostrar que ha asimilado todo lo que le han enseñado durante estas últimas semanas y que domina las técnicas básicas de programación aplicadas al lenguaje Java. Tipos de datos, literales, identificadores, operadores y expresiones, son conceptos con los que va a trabajar de ahora en adelante y que debe dominar para llegar a ser un buen programador de ordenadores.*

En esta unidad vas a darte cuenta de que no aparecen grandes conceptos nuevos. La mayoría de las cosas que te vamos a contar ya las hemos visto de forma general, para cualquier lenguaje, en las unidades 2 y 3, dedicadas a Tipos de datos, a Operadores, Expresiones e Instrucciones. Incluso ya incluíamos allí algunos comentarios y ejemplos que hacían referencia a Java.

En esta unidad trataremos de concretar algunos detalles del lenguaje Java, y de recordar lo que aprendimos en esas unidades anteriores. Además, se trata de que a través de ejemplos, puedas probar tú mismo el funcionamiento de los operadores, la construcción de expresiones, el uso de identificadores, comprobando con el compilador que lo que has escrito no sólo no tiene errores, si no que ¡además funciona!

Unidad Didáctica VIII

Literales en Java

*El último trabajo llegado a **SI Andalucía**, es la actualización de una aplicación para la gestión de nóminas de una empresa, que necesita una serie de adaptaciones a la nueva normativa y además han decidido añadirle algunas mejoras para obtener determinados informes que pueden ser útiles para el funcionamiento de esa empresa.*

*Este trabajo se le ha asignado a **Carmen** que conoce bien el tema y **José** ha pensado que podría ser una buena oportunidad para ver qué tal se defiende **Víctor** como programador en Java. Cree que es ideal que trabajen juntos en esta aplicación. Le dice a **Carmen** que lo ideal sería que dejara a **Víctor** iniciar la preparación de la programación decidiendo las variables a utilizar y los tipos de cada dato.*

Recuerda que en la unidad 3, dedicada a "Operadores, Expresiones e Instrucciones" se introducía el concepto de literal. Los literales **son valores concretos para un tipo de datos básico del lenguaje.**

En el caso de Java tendremos que conocer los diferentes tipos de literales para los distintos tipos de datos básicos de este lenguaje. Y aunque esos tipos básicos ya se mencionaban como ejemplos en la unidad 2, dedicada a "Datos: Tipos y Características", conviene repasarlos aquí, junto a las reglas de formación de literales de cada tipo.

La lista de tipos primitivos en Java es la siguiente:

- boolean
- char
- byte
- short
- int
- long
- float
- double

Autoevaluación

De los siguientes, ¿cuál no es un tipo básico en Java? Señala la afirmación correcta

- a) short.
- b) float.
- c) double.
- d) Todas las anteriores son literales en Java.

Comprobar

Unidad Didáctica VIII

Literales para el tipo boolean

¿Recuerdas en qué consistía el tipo boolean? Representa datos de tipo lógico, es decir, que pueden tomar sólo los valores **verdadero o falso**.

Recuerda que cuando evaluamos cualquier expresión lógica en un programa, del tipo `edadTrabajador = = 37`, el resultado será verdadero si la variable realmente tiene el valor 37 y falso si es distinto de 37. No hay más posibilidades. Pues bien, esos valores (verdadero o falso) son todos los posibles para el tipo boolean, y se representan por medio de los siguientes dos literales:

- El valor verdadero con el literal `true`.
- El valor falso con el literal `false`.

Aunque **true** y **false** son literales, a efectos prácticos se comportan casi como palabras reservadas del lenguajes, ya que no los podremos usar para otro fin distinto del que el lenguaje les asigna.

Un ejemplo de su uso:

Víctor tiene que almacenar de alguna forma información relativa al estado civil de los empleados, ya que la empresa asigna un complemento especial para trabajadores casados, con el fin de facilitar la conciliación de la vida familiar y laboral.
Víctor decide que esa información se va a guardar para cada empleado en una variable de tipo boolean llamada casado.

```
boolean casado;
```

¿Cómo indicar que un empleado está casado?

```
casado = true;
```

¿Cómo indicar que un empleado está soltero, separado o viudo?

```
casado = false;
```

¿Cómo comprobar si hay que asignarle un complemento a un empleado?

```
if(casado)
        sueldo=sueldo+complementoCasado;
```

Observa que la variable de tipo lógico es por sí misma una expresión de tipo lógico, es decir una condición. Por eso **no es necesario escribir**:

```
if(casado==true)
```

Autoevaluación

Suponiendo una variable de tipo boolean llamada jubilado que indica si un trabajador está jubilado o en activo.¿Cómo escribirías en Java que para los trabajadores que no estén jubilados su declaración es igual al sueldo bruto menos las retenciones?.Señala la afirmación correcta

a) `if (jubilado) declaracion=sueldobruto-retenciones;`
b) `if (jubilado == false)`

- `declaracion=sueldobruto+retenciones;`
- c) `if (jubilado == false) declaracion=sueldobruto-retenciones;`
- d) `if (jubilado == true) declaracion=sueldobruto-retenciones;`

Comprobar

De las siguientes afirmaciones respecto a los literales de tipo boolean, señala la afirmación correcta.

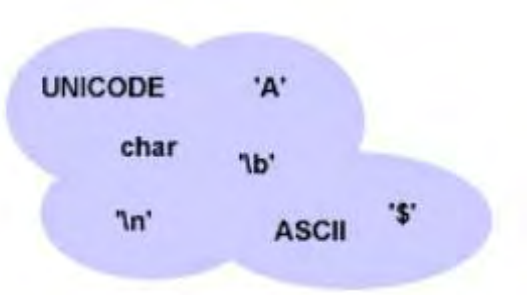
- a) Representa datos de tipo condicional.
- b) Pueden tomar cualquier valor siempre y cuando sean distintos dos a dos.
- c) Se utilizan exclusivamente para comprobar el valor de una condición.
- d) Todas las anteriores son falsas.

Comprobar

Unidad Didáctica VIII

Literales para el tipo char

¿Recuerdas qué tipo de datos representa el tipo **char**? Representa al conjunto de caracteres del alfabeto usado por el lenguaje. En el caso de Java, **los literales de tipo char son cada uno de los caracteres del alfabeto que usa el lenguaje Java**. Recordamos que el alfabeto usado por Java es el alfabeto **Unicode**, que gracias a los 16 bits que usa para representar cada carácter, permite representar una amplísima variedad de símbolos o caracteres (en total 2^{16} = 65.536 símbolos distintos).



Abarca la práctica totalidad de símbolos usados por las lenguas existentes en el mundo, así como símbolos científicos y caracteres de control. ¿Cómo representamos cada uno de esos caracteres?

Los literales de caracteres se representan mediante comillas simples. Cualquier símbolo Unicode situado entre comillas simples es un literal de tipo char.

Así, Víctor ha asignado una variable letraNIF para cada empleado que contendrá la letra del NIF del trabajador.

```
char letraNIF;
```

¿Cómo asignarle un valor?

```
letraNIF='E';
```

Otros ejemplos de literales de tipo char son:

'a', '0', '9', 'Z', '? ', ') ', '+', '$', '€', 'A', ' ', (espacio en blanco), '' (dos comillas simples, usadas para representar el carácter nulo, que no es lo mismo que el espacio en blanco y que tampoco es lo mismo que el carácter comillas dobles)

No obstante, existen muchos caracteres Unicode, y no todos los podemos escribir directamente con nuestros teclados de 102 teclas. ¿Para qué queremos tantos caracteres Unicode si no los vamos a poder escribir con nuestro teclado?

Justamente para poder representarlos todos, **el lenguaje Java proporciona varias formas alternativas de representar los literales de tipo char.**

- La primera de ellas es representarlos **mediante secuencias de escape, usando la contrabarra (\) seguida de algún carácter que debe interpretarse de forma especial** . Lo que escribimos tras la contrabarra se interpreta como un código para representar un carácter:
  '**\b**' Espacio hacia atrás '**\u0008**'
  '**\n**' Nueva línea '**\u000A**'
  '**\t**' Tabulador '**\u0009**'
  '**\"**' La comilla doble '**\u0022**'
  '\'' La comilla simple '**\u0027**'
  '****' La contrabarra **\u005C**'
- También podemos representar un literal de tipo char **mediante su código en hexadecimal al que le añadimos delante el carácter de escape \u (de unicode)**. En los ejemplos anteriores hemos representado esta posibilidad para los caracteres en cuestión. Cada símbolo hexadecimal representa a una combinación de 4 bits, por lo que necesitamos 4 símbolos hexadecimales para representar cada carácter Unicote de 16 bits.
- Pero el código Unicode es compatible con el código ASCII, ya que para los caracteres ASCII, el código Unicode es igual que el ASCII, completando con 8 ceros los bits más a la izquierda del código. Pues también podemos representar **los caracteres ASCII mediante su código en octal tras el**

**carácter de escape **

Así, la letra B es el carácter 66 tanto del código ASCII como del código Unicode. Podríamos representarla de las siguientes formas:

- 'B' (directamente, escribiendo el carácter entre comillas simples)
- '\u0042' (con su código hexadecimal, ya que 66 en hexadecimal es 42, y expresado con 4 posiciones hexadecimales es 0042)
- '\102' (con su código octal, ya que 66 en octal es 102)

También podremos representar cualquier carácter Unicode. Por ejemplo, la letra Pi mayúscula del alfabeto griego, es el carácter Unicode 0370 en hexadecimal. Por tanto, '\u0370' es el literal de tipo char Pi mayúscula (Π)

Autoevaluación

De las siguientes formas alternativas que nos proporciona el Java de representar los literales de tipo char, señala la afirmación correcta:

a) Se pueden representar mediante secuencias de escape, usando la contrabarra (\) seguida de algún carácter que debe interpretarse de forma especial.

b) Se pueden representar mediante su código en hexadecimal al que le añadimos delante el carácter de escape \u (de unicode).

c) Pues también podemos representar los caracteres ASCII mediante su código en octal tras el carácter de escape \

d) Todas las anteriores son correctas

Comprobar

Unidad Didáctica VIII

Literales para los tipos enteros

Recuerda que en la unidad 2 hablábamos de los datos de tipo entero y veíamos que Java tenía 4 tipos de datos enteros. Casi cualquier aplicación tendrá necesidad de usar números enteros para contar, y representar cantidades.

Esos tipos son los que se muestran en la siguiente tabla, que ya describimos en la unidad 2:

Nombre del tipo entero	Tamaño usado para su representación (bits)	Total de números distintos representables	Menor número representable para el tipo	Mayor número representable para el tipo
byte	8 =1 byte	$2^8 = 256$	-128	+127
short	16 =2 bytes	$2^{16} = 65.536$	-32.768	+32.767
int	32 = 4 bytes	$2^{32} = 4.294.967.296$	-2.147.483.648	+2.147.483.647
long	64 = 8 bytes	$2^{64} = 1,844674407 E+19$	-9.223.372.036.854.775.808	9.223.372.036.854.775.807

Las diferencias entre ellos son fundamente el rango de valores que permiten ocupar, y que depende de la cantidad de bits que se reservan para las variables de cada tipo. Pero dentro de cada rango, **la forma de representar los literales de cada tipo es similar**.

Un literal para un tipo entero se forma como una secuencia de dígitos (0-9).

Debemos tener en cuenta que para cada tipo primitivo de enteros (byte, short, int y long) se deben utilizar valores comprendidos en su rango.

Los números enteros se pueden especificar en tres notaciones: decimal, octal y hexadecimal.

- Por defecto, todos los números se consideran que están escritos en base decimal
- Los números que comienzan por un 0 (cero), se consideran que están en base octal
- Los números que comienzan por 0x o 0X (un cero seguido de una x, mayúscula o minúscula) se tratan como números en hexadecimal. En este caso podrán contener, además de dígitos del 0 al 9, las letras A, B, C, D, E o F (en mayúscula o minúscula).

Así, por ejemplo, los siguientes literales representan todos ellos el mismo valor numérico de tipo entero:

Decimal							
459							
Octal							
0713							
Hexadecimal							
0X1CB	0x1CB	0X1cb	0x1cb	0x1Cb	0x1cB	0X1cB	0X1Cb

Pero como hemos comentado en más de una ocasión, **en las aplicaciones de gestión rara vez tendremos necesidad de usar otra representación que no sea la asumida por defecto**, y que consiste en escribir sin más el número a representar, tal y como lo hacemos en la vida diaria, y usando por supuesto el sistema decimal. Otras representaciones sólo aportarían confusión a la mayoría de nuestros programas. En el ejemplo, el primer literal es el que usarás seguramente siempre para representar al número decimal 459.

Claro y conciso.

Compilando y ejecutando el siguiente programa comprobarás que efectivamente, al escribirlos en la pantalla, todos los literales generan el mismo valor entero.

DEMO: Observa cómo se establece el PATH, cómo se compila y se ejecuta un archivo de Java

Descarga el archivo LiteralesEnteros.java

Autoevaluación

De las siguientes afirmaciones respecto a los literales de tipo entero, señala la afirmación correcta.

- a) Un literal para un tipo entero se forma como una secuencia de dígitos (0-9).
- b) Los números enteros se pueden especificar en tres notaciones: decimal, octal y hexadecimal.
- c) Los números que comienzan por un 0 (cero), se consideran que están en base octal.
- d) Todas las anteriores son correctas

Comprobar

Respecto a los tipos de enteros podemos afirmar:

- a) El tamaño utilizado para representar los enteros denominados short es de 32 bits.
- b) El tamaño utilizado para representar los enteros denominados int es de 16 bits.
- c) El rango de valores del tipo entero de Java denominado short está comprendido entre -32.768 y +32.767.
- d) Todas las anteriores son correctas.

Comprobar

Respecto a los tipos de enteros podemos afirmar:

- a) El tamaño utilizado para representar los enteros denominados long es de 8 bytes.
- b) El tamaño utilizado para representar los enteros denominados int es de 32 bits.
- c) El rango de valores del tipo entero de Java denominado byte está comprendido entre -128 y +127.
- d) Todas las anteriores son correctas.

Comprobar

Unidad Didáctica VIII

Ejemplos con literales enteros. LiteralesEnterosErrorDeTipo.java

Hemos visto como podemos escribir los literales de tipo entero, pero aún no estamos preparados para entender algunos aspectos de su uso.

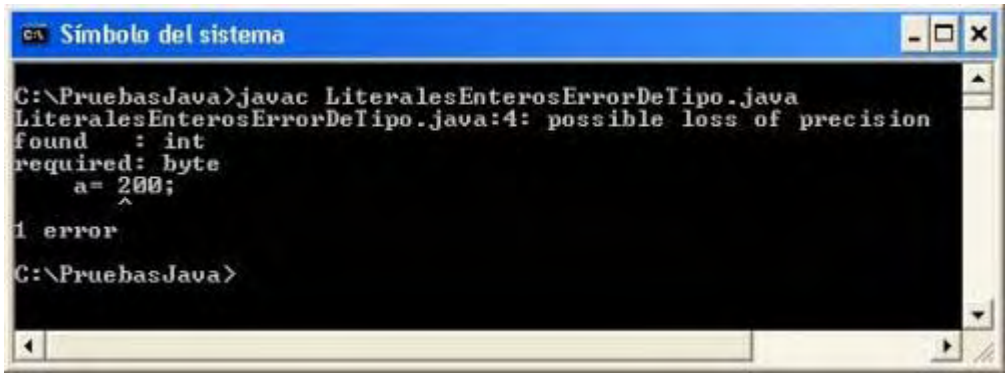
Vamos a introducirlos con un par de ejemplos.

El primer ejemplo que te mostramos incluye deliberadamente un error de compilación. Se debe a que estamos intentando asignarle a una variable de tipo byte un literal que está fuera del rango representable para ese tipo. Compílalo, ejecútalo, y lee el error que te proporciona el compilador. Los comentarios que se hacen posteriormente te terminarán de aclarar lo que sucede.

DEMO: Observa la ejecución del archivo y el error

Descarga el archivo LiteralesEnterosErrorDeTipo.java

El error que da el compilador lo puedes ver a continuación:



Lamentablemente, el compilador ofrece los mensajes en Inglés, porque como la mayoría de los lenguajes, ha sido diseñado por ingenieros de países del ámbito anglosajón. Pero no debes preocuparte si tu dominio del inglés no es demasiado bueno. La mayoría de los mensajes no requieren grandes conocimientos de inglés para poder traducirlos, y suelen repetirse frecuentemente, por lo que acabarás identificándolos sin problemas. De todas formas, un buen nivel de inglés ayuda bastante a la hora de consultar documentación sobre el lenguaje.

*Tanto **Víctor** como **Carmen** han decidido apuntarse a un curso de inglés técnico. Aunque para **Carmen** su desconocimiento de este idioma no fue un impedimento para conseguir su título de Técnico Superior en Desarrollo de Aplicaciones Informáticas, reconoce que un mayor conocimiento de la lengua inglesa le hubiera facilitado las cosas. Está convencida que le será de gran ayuda en su ámbito laboral, donde frecuentemente tiene que consultar documentación técnica en inglés.*

Siguiendo con nuestro ejemplo, el mensaje de error viene a decir:

... : 4: posible pérdida de precisión.

encontrado: int

requerido: byte

...

Además se marca el comienzo del literal 200 como causa del problema. **¿A qué se debe?**

200 es un valor que excede el límite máximo representable con los 8 bits del tipo byte. Ese valor máximo es 127. Por tanto el compilador nos dice que si intento meter un literal de tipo int (de 32 bits) en una variable de tipo byte (de 8 bits) es posible que se pierda información. Lo cual no deja de ser bastante lógico y evidente.

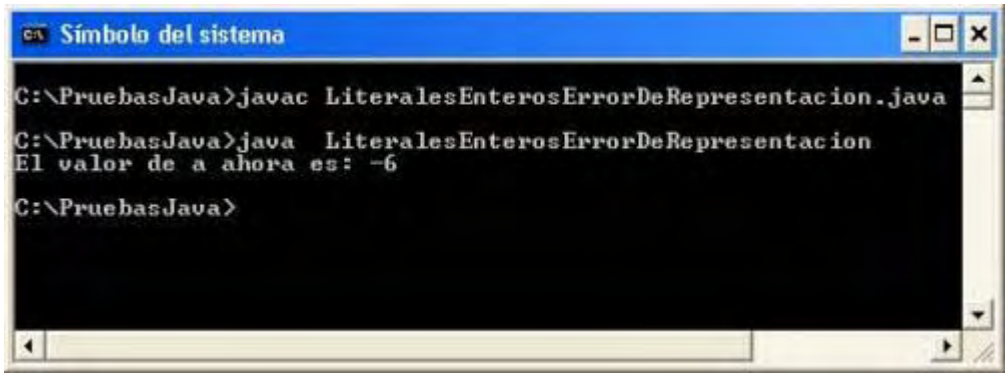
Unidad Didáctica VIII

Ejemplos con literales enteros. LiteralesEnterosErrorDeRepresentacion.java

El segundo ejemplo muestra que los números enteros en Java tienen una **aritmética circular**. Si al sumar dos números enteros válidos obtenemos un resultado que excede el rango permitido para el tipo en cuestión, e intentamos asignarlo a una variable de ese tipo forzando la conversión del resultado a ese tipo, el compilador no nos proporciona ningún error, y hace la operación, pero el resultado obtenido puede ser desconcertante. Así, si sumamos 125 consigo mismo, en una variable de tipo byte, y forzamos la conversión del resultado a byte mediante un casting explícito para que no ocurra el error del ejemplo anterior, el resultado que obtenemos es -6, como podemos ver en el ejemplo.

DEMO: Compila y observa el resultado que obtenemos

Descarga el archivo LiteralesEnterosErrorDeRepresentacion.java



Lo que ocurre es que si al mayor número representable para un tipo entero (en nuestro ejemplo 127 para byte) le sumamos 1 obtenemos el menor número representable para ese tipo (-128 para byte)

Si a 125 le sumamos 125 (125=3+122) realmente al sumarle 3 pasamos al menor número representable, que es -128, y al sumarle de nuevo 122, obtenemos el -6 del resultado. (- 128 + 122 = - 6)

Es importante tener en cuenta estos detalles al programar, porque si no manejamos bien estas situaciones, el resultado del programa puede ser bien distinto de lo que esperamos.

Como norma general, se aconseja usar siempre int para los números enteros, aunque sepamos que van a almacenar datos pequeños, para reducir las posibilidades de que esos errores ocurran.

Unidad Didáctica VIII

Literales para los tipos reales

¿Recuerdas qué tipo de datos representábamos como reales? **Eran básicamente los números con cifras decimales,** y teníamos varios tipos de datos reales, con más o menos precisión según el número de bits usados para representarlos. Concretamente en Java tenemos dos tipos de datos reales, que son **float y double**. Mira esta tabla para recordar las diferencias entre ambos.

Nombre del tipo real	Tamaño usado para su representación (bits)	Total de números distintos representables	Menor número representable para ese tipo (en valor absoluto)	Mayor número representable para ese tipo (en valor absoluto)
float	**32 = 4 bytes**	**2^{32} = 4.294.967.296**	**1.401298464324817 E-45**	**3.4028234663852886 E+38**
double	**64 = 8 bytes**	**2^{64} = 1,844674407 E+19**	**4.9 E-324**	**1.7976931348623157 E+308**

Pero, ¿cómo formamos los literales de tipo real?

Un literal para un número se considera de tipo real si cumple alguno de los siguientes requisitos:

- **Si posee decimales** (se utiliza el punto como separación entre la parte entera y la decimal) 124.85
  - **Tanto la parte entera como la parte decimal puede estar vacía**, es decir, un literal real puede no tener parte decimal o no tener parte entera, en cuyo caso se supondrá que es cero. Así 124. equivale a 124.0 y .85 equivale a 0.85
- **Si posee un exponente** (el exponente se introduce mediante la letra e ó E) 124850.0 equivale a:
  - 12485E1(aunque no contenga punto decimal, es real, por incluir la letra e de exponente)
  - 12485e1
  - 12485e+1
  - 124.85e3
  - 1248500E-1
  - .12485e6
  - 1.248500E5
  - etc.
- **Si va seguido por una letra identificativa de tipo real**
  - f ó F para **float** 28f , 28F y 28.0f son literales de tipo float.
  - d ó D para **double** 28d , 28D , 28.0d y 28.0 son literales de tipo double.

Es importante tener en cuenta que **en caso de no especificar la letra correspondiente al tipo float, todos los números reales se consideran en Java de tipo double por defecto**. Como vimos en la unidad 2, es preferible usar el tipo de máxima precisión en los cálculos para minimizar errores, y el mayor tamaño necesario para el tipo double, no suele ser un inconveniente en realidad si tenemos en cuenta el tamaño de las memorias de los ordenadores actuales.

Así, como hemos indicado antes, 28.0 será considerado como un literal de tipo double.

Unidad Didáctica VIII

Diferencias al operar con literales enteros o reales

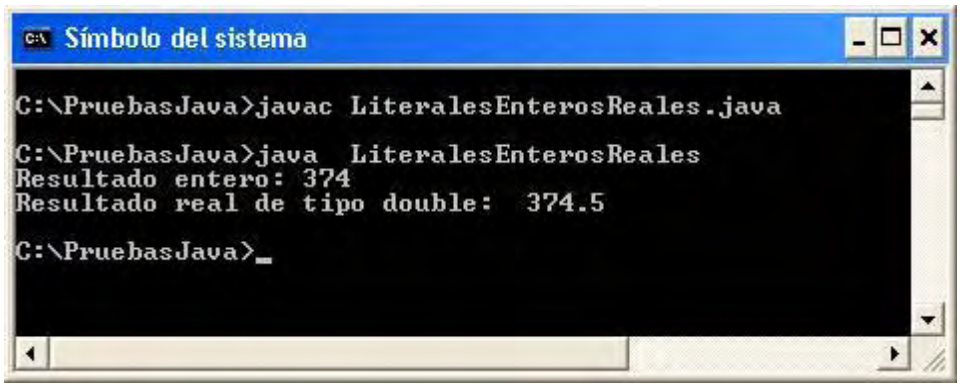
A estas alturas debes saber cómo expresar en cualquier programa Java los literales de los distintos tipos. Pero aún falta un detalle. Hay operadores que funcionarán de forma distinta si los literales son de un tipo o de otro. En el ejemplo que te proponemos puedes apreciar la diferencia.

DEMO: Compila y observa el resultado

Descarga el archivo LiteralesEnterosReales.java

Al compilarlo y ejecutarlo, habrás comprobado que a pesar de que la operación era la misma, y los operandos "iguales", el resultado obtenido es distinto. La única diferencia es que el dividendo es int en un caso y double en otro.

El operador de división está sobrecargado. La sobrecarga de operadores es una característica de Java que permite que el mismo operador haga operaciones distintas dependiendo del tipo de los operandos. Así, el operador \ es la división entera (sin sacar decimales) si los dos operandos son números enteros y será la división real (sacando decimales) si alguno de los operandos es de tipo real.



Como puedes apreciar, en un caso el resultado es de tipo entero, sin decimales (374) y en el otro caso es real, con decimales (374.5). Eso se debe a que 3745d es un literal de tipo double, aunque aparentemente no lleve decimales, y por tanto se hace la división real. Internamente, el compilador convierte el segundo operando 10 en 10.0, y realiza la división real.

Unidad Didáctica VIII

Identificadores en Java

__Víctor__ tiene claras cuales serán las variables del programa, pero al mostrárselas a __José__, éste no está de acuerdo con los identificadores (nombre) que ha utilizado para cada una de ellas, dice que no son suficientemente descriptivos con respecto al contenido de la variable. La verdad es que el propio __Víctor__ entiende la postura de su compañero cuando le explica que N1, N2 y N3, no son nombres de variables adecuados cuando se trata de recoger los valores del número de horas extras trabajadas, número de horas de ausencia del trabajador y número de horas semanales del trabajador según convenio. José le pide que utilice identificadores similares a los que le propone:

- *horasExtrasTrab.*
- *horasAusencia.*
- *horasSemanalesConvenio.*

__José__ le explica que debe tener claro el uso que se le dará a cada variable y para eso al ver su identificador debes asociarla inmediatamente a su contenido. Esto va a ser especialmente útil al usar esas variables en fórmulas o expresiones aritméticas.

¿Recuerdas lo que es un identificador? En la unidad 3 definíamos a **los identificadores como los nombres que el programador decide asignar a los elementos que crea en su programa**. Ya estudiamos allí el concepto, y vimos que cada identificador está asociado a una zona de memoria. También se vio la necesidad de usar identificadores claros y descriptivos, pero no excesivamente largos.

¿Podrá Víctor, con su conocida creatividad, llamar a una variable ZaXxOn23?

Aquí se trata de conocer las reglas que fija el lenguaje Java para construir identificadores.

Un identificador es una secuencia de **uno o más símbolos Unicode** que cumple las siguientes condiciones:

- Puede tener cualquier longitud, **no hay tamaño máximo**.
- **El primer símbolo de la secuencia es una letra, un símbolo de subrayado (_) o un símbolo dólar ($).**
- El resto de caracteres de la secuencia pueden ser letras o dígitos mezclados indistintamente.

Cabe destacar:

1. **Se puede usar cualquier símbolo Unicode**, lo cual significa que por ejemplo, un programador japonés podrá ponerle nombres a sus variables, y demás elementos del programa, usando símbolos del alfabeto japonés, siempre y cuando su entorno de trabajo permitiera escribir Unicode (concretamente caracteres japoneses) directamente. Lo que no se puede usar es una secuencia de escape para representar ese carácter japonés con su código, ya que no pueden empezar los caracteres con \.
2. **No hay límite de tamaño** para un identificador, pero la lógica impone usar nombres no excesivamente largos, que resulten difíciles de escribir y manejar, ni tan cortos que no nos permitan identificar con claridad el uso que se hace del componente al que se le quiere dar nombre.
3. **No se pueden usar espacios** en blanco en medio de un identificador. Es un separador y realmente le estaría proporcionando al compilador dos identificadores en vez de uno.
4. Aunque no es obligado su cumplimiento, **debe seguirse el convenio** para nombrar identificadores en Java:
    1. Variables, constantes de objeto, métodos, etc: La primera palabra del identificador en minúscula, y sólo la primera letra de cada una de las siguientes palabras en Mayúsculas. Ej: `nombreDelEmpleado`
    2. Clases e interfaces: Igual, pero la primera letra de la primera palabra, también en mayúscula. Ej: `EmpleadoEmpresa`
    3. Constantes de clase: Todas las letras en mayúsculas, y

las palabras separadas por el símbolo de subrayado. Ej:
`MAXIMO_VALOR_PERMITIDO`

5. Aunque existen caracteres no permitidos, lo mejor es usar nombres razonables, y si el compilador en algún momento nos indica que hay un error en el identificador, lo corregimos.
6. **Las palabras reservadas no pueden usarse como identificadores**. Ej: no puedo tener una variable que se llame **int**.

PARA SABER MÁS
Lee los siguientes enlaces para conocer más sobre las clases en Java. El primero es un sitio en el que puedes acceder a un manual en el que empieza por explicar qué son las clases y cómo utilizarlas.
Clases en Java [Versión en caché]
En este otro enlace puedes ver más sobre clases en Java y cómo gestionar archivos ".java" donde guardar clases públicas.
Ficheros .java [Versión en caché]

Autoevaluación

Respecto a los identificadores de Java, señala la afirmación correcta

- a) Las palabras reservadas no pueden usarse como identificadores.
- b) Se puede usar cualquier símbolo ASCII.
- c) Se pueden usar espacios en blanco en medio de un identificador.
- d) Todas las anteriores son correctas.

Comprobar

Un identificador es una secuencia de **uno o más símbolos Unicode** que cumple las siguientes condiciones. Señala la afirmación correcta

- a) El resto de caracteres de la secuencia pueden ser letras o dígitos mezclados indistintamente.
- b) Puede tener cualquier longitud, **no hay tamaño máximo**.
- c) El primer símbolo de la secuencia es una letra, un símbolo de subrayado (_) o un símbolo dólar ($).
- d) Todas las anteriores son correctas.

Comprobar

Unidad Didáctica VIII

Ejemplo de errores con identificadores

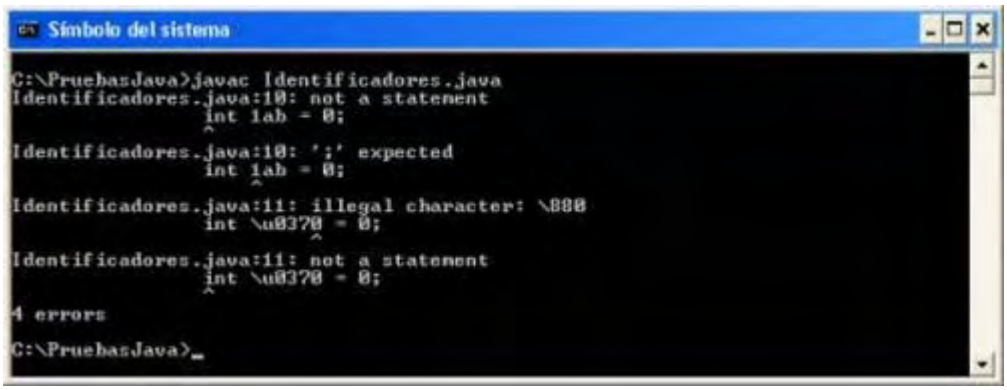
¿Serán frecuentes los errores al formar identificadores? ¿Son muy estrictas las normas de formación de identificadores? La realidad es que no. Podemos llamar casi de cualquier manera a casi cualquier cosa, siempre y cuando no tengamos un especial gusto por los nombres raros y rebuscados.

En el siguiente ejemplo, se incluyen algunos identificadores raros, algunos de ellos correctos y otros deliberadamente incorrectos. Afortunadamente el compilador nos avisa de lo que pasa, pero es conveniente analizar los mensajes de error que nos envía, ya que no todos son claros.

DEMO: Observa los errores

Descarga el archivo Identificadores.java

Al compilar el ejemplo, obtendrás el siguiente listado de errores:



Los dos primeros errores están asociados a la misma línea de código: `int 1ab = 0;`

- Después de la palabra reservada int se espera un identificador. Como lo que se encuentra empieza con el número 1, no es un identificador válido, e indica que no es una sentencia válida. (**`not a statement`**).
- A continuación intenta "recuperarse del error", y hace como si tras la palabra int hubiera un identificador válido para formar una sentencia de declaración, como si el programador hubiera olvidado ponerlo, e intenta encontrarle sentido a lo que viene detrás suponiendo que ese identificador estuviera. Al encontrarse de nuevo con el 1, entiende que la sentencia de declaración ha terminado, y que por tanto debe aparecer un punto y coma. (**`';' expected`**).
- A continuación encuentra un carácter no válido para un identificador, ya que éstos no pueden empezar por '\' (**`illegal character: \880`**) y eso hace que toda la sentencia no sea válida (**`not a statement`**)

Ninguno de los dos primeros errores indica claramente el problema real, que es el uso de un carácter de comienzo erróneo para un identificador. Debemos acostumbrarnos a que **el compilador no siempre nos muestra el error exacto que se ha producido, ni siquiera el lugar exacto**. Al intentar recuperarse del error y seguir compilando, debe hacer suposiciones del tipo de las que hemos mencionado, que no siempre son acertadas.

Por eso, lo normal es:

- **Revisar sólo los primeros errores tras una compilación.**
- **Ver los que son claramente identificables y corregirlos.**
- **En el momento en que los errores nos resulten extraños y poco claros, compilar de nuevo**.

- **Es posible que un listado de cientos de errores de compilación se deba realmente a un único error**, que al ser corregido hace que desaparezcan todos los demás.

Saber interpretar esos mensajes de error es una tarea que lleva tiempo y que exige mucha práctica y conocimiento del lenguaje. Además, después de haber compilado muchos programas, habrá muchas situaciones similares que se repetirán, y que ya no nos costará tanto trabajo identificar.

Unidad Didáctica VIII

Operadores en Java

__Víctor__ tiene casi todo el trabajo hecho, y ahora le corresponde a __Carmen__ programar las operaciones a realizar sobre los datos, para ello decide escribir en Java todas las fórmulas a usar, para intentar posteriormente aplicarles las particularidades de cada caso.

__Víctor__ observa que esta parte no podría haberla programado él porque no conoce a fondo los operadores en Java y decide ponerse manos a la obra, para que en la próxima aplicación no necesite la ayuda de su compañera. __Carmen__ tiene que agradecer a __José__ la sugerencia que hizo a __Víctor__ sobre cómo nombrar los identificadores de las variables, ahora su trabajo resulta un poco más sencillo.

¿Recuerdas la función que tienen los operadores en un lenguaje? Los literales y los identificadores son las piezas básicas del lenguaje, pero hay que combinarlos para formar expresiones que aporten significado a las sentencias o instrucciones, de la misma manera que las palabras se combinan para construir sintagmas que aportan significado a cada frase. ¿Cómo combinábamos los literales y los identificadores para formar expresiones? La respuesta es que por medio de los operadores.

Todo eso ya lo sabes de la unidad 3. El objetivo en esta unidad es recordártelo y probar algunos ejemplos de uso de los distintos operadores que hay en Java.

Es el momento de recordar la lista de operadores de Java.

Unidad Didáctica VIII

Operadores Aritméticos

Suma	+	Operador binario (necesita dos operandos)
Número positivo o signo	+	Operador monario o unario (necesita un operando)
Resta	-	Operador binario
Número negativo o signo	-	Operador monario o unario
Multiplicación	*	Operador binario
División	/	Operador binario
Resto-Módulo	%	Operador binario

Vamos a ver un ejemplo en Java de uso de los distintos operadores aritméticos, para que puedas ejecutarlo.

Carmen, en el programa de nóminas, se encuentra con la necesidad de resolver el problema siguiente:

- Para la empresa trabajan autónomos, que intervienen en tareas concretas cuando son necesarios, y durante el tiempo que son necesarios.
- La mayoría de estos trabajadores trabajan días sueltos, o por temporadas, cuando hay más trabajo. Sin embargo, la empresa calcula el pago por semanas completas, considerando que cada 5 días sueltos, constituyen una semana de sueldo, sin contar el fin de semana.
- Cada 4 semanas (aproximadamente un mes), se liquida con ellos, de forma que cada semana se paga a la cuarta parte del sueldo base mensual correspondiente a la categoría profesional del trabajador.
- Los días sueltos que queden, que no lleguen a completar una semana se pagan algo más caros, para compensar los desplazamientos necesarios para poco trabajo, y en vez de pagarse proporcionalmente a los 30 días de un mes, se pagan proporcionalmente a razón de la 20ª parte del sueldo base mensual (con 20 días sueltos, se ganaría el salario de un mes).
- Al sueldo bruto así calculado se le aplica la retención del IRPF, que para este tipo de trabajadores, con carácter general es el 15%.
- Los datos los debe introducir en el programa a partir de unos partes de trabajo en los que los trabajadores anotan los días trabajados en cada semana.
- Hay que tener en cuenta que el sueldo se paga en Euros, por lo que debe admitir decimales.
- La salida del programa debe ser un mensaje en pantalla en el que indique el neto total a pagarle al trabajador, para que el cajero sepa cuanto debe pagarle, y el importe retenido, para informar al trabajador.

Un programa básico que resolvería este problema usando los operadores aritméticos puede ser el del enlace siguiente. Aparecen dos enlaces porque necesitaremos nuestra clase ES para realizar la entrada de datos desde teclado.

Recuerda: El fichero ES.java debe estar en la misma carpeta o directorio que AritmeticaNominas.java para que se compile fácilmente.

DEMO: Observa el funcionamiento del programa de nóminas

☐ **Descarga el archivo AritmeticaNominas.java**

☐ **Descarga el archivo ES.java**

Unidad Didáctica VIII

Operadores Relacionales

>	**Mayor que**
>=	**Mayor o igual que**
<	**Menor que**
<=	**Menor o igual que**
= =	**Igual**
!=	**Distinto (no igual)**

Tras haber realizado la pequeña aplicación anterior, el usuario de la aplicación de nóminas le dice a Carmen que también debe tener en cuenta el cálculo de un complemento o de una retención adicional, según los casos, en la nómina.

- Si el sueldo neto de un mes de un trabajador autónomo, tal y como se había calculado en el ejemplo anterior, es menor o igual al salario mínimo, el complemento será del 5% de su sueldo neto previo al cálculo del complemento.
- Suponemos que el salario mínimo es un dato que se proporciona por teclado.
- Se trata de un complemento exento de IRPF.
- Ese complemento será sumado al sueldo neto, cuando proceda aplicarlo, para mostrar en pantalla el importe de ese complemento junto al sueldo neto final, y la retención por IRPF.
- Si por el contrario su sueldo neto es superior al salario mínimo en 600 € ó más, se le incrementará en un 2% la retención por IRPF a aplicar sobre el sueldo neto sin incluir complemento.
- Igualmente se mostrará en la pantalla el sueldo neto final tras aplicar ese descuento adicional, junto al IRPF descontado y al importe del complemento correspondiente. Carmen se pone a trabajar y elabora una nueva versión del programa, usando algunos operadores relacionales para tener en cuenta esas situaciones.

DEMO: Observa el programa de nóminas modificado en funcionamiento

Descarga el archivo AritmeticaNominasRevisada1.java

Recuerda que a partir de ahora, cada vez que quieras realizar alguna operación que implique una lectura de datos desde teclado, tendrás que incluir la clase ES, para que todo funcione correctamente de forma sencilla.

Unidad Didáctica VIII

Operadores Lógicos

!	**Negación (not)**
&&	**Conjunción (and)**
||	**Disyunción (or)**

Cuando Carmen le entrega la versión corregida del programa al empleado de la empresa que va a ser el usuario, éste le dice que se ha acordado de otro detalle. Carmen se arma de paciencia. Sabe que es habitual que el cliente se acuerde de los detalles cuando el programa está terminado, así que esto forma parte del proceso normal de depuración de errores, y escucha las peticiones del empleado.

Éste le dice que el complemento es sólo para trabajadores casados, que cumplan unos requisitos:

- Si el trabajador eventual no está casado no le corresponderá complemento alguno o
- si su sueldo bruto es mayor del doble del salario mínimo sin tener hijos, tampoco le corresponderá complemento alguno.

Por tanto, Carmen debe modificar el programa para incluir una condición que compruebe antes de calcular el complemento si procede hacer el cálculo o no.

Debe cumplirse:

a. Que el trabajador esté casado y
b. Al menos una de las dos condiciones siguientes
   - Que su sueldo bruto sea menor o igual que el doble del salario mínimo o bien
   - Que tenga hijos (que su número de hijos sea distinto de cero)

En Java esa condición se expresa como así:

```
casado && (sueldoBruto <= salarioMinimo * 2 || nºHijos != 0)
```

Y su funcionamiento lo puedes comprobar en la segunda revisión de la aplicación AritméticaNominas, a la que accederás a través del siguiente enlace.

DEMO: Observa la ejecución del programa de nóminas modificado otra vez

Descarga el archivo AritmeticaNominasRevisada2.java

Autoevaluación

Indique el valor de las siguientes expresiones (en Java), suponiendo a y b variables de tipo lógico:

- a) a=verdadero , b=falso ; **a || b** es FALSO.
- b) a=verdadero , b=falso ; **a && b** es VERDADERO.
- c) a=falso , b=falso ; **a || b** es VERDADERO .
- d) a=verdadero , b=falso ; **a || b** es VERDADERO.

Comprobar

Indique el valor de las siguientes expresiones (en Java), suponiendo a y b variables de tipo lógico:

a) a=verdadero , b=falso; **a && !b** es VERDADERO.
b) a=falso , b=falso; **a || !b** es VERDADERO.
c) a=falso , b=verdadero; **a || b** es VERDADERO. .
d) Todas las anteriores son ciertas.

Comprobar

Indique el valor de las siguientes expresiones (en Java), suponiendo a y b variables de tipo lógico:

a) a=verdadero , b=falso ;**a && b** es VERDADERO.
b) a=verdadero , b=verdadero ;**a || b** es FALSO.
c) a=falso, b=verdadero ; **!a && b** es VERDADERO.
d) Todas las anteriores son ciertas.

Comprobar

Unidad Didáctica VIII

Operadores de Asignación

Asignación = , += , -= , *= , /= , %= , &= , |= , ^= , <<= , >>= , >>>=

Realmente es difícil escribir un programa sin que se utilicen algunas sentencias de asignación, y en los ejemplos anteriores hay numerosas sentencias que incluyen al operador de asignación básico (=).

Aquí vamos a ver algunas de ellas, destacando el hecho de que pueden simplificarse si usamos alguno de los operadores de asignación combinados. El operador de asignación básico, como ya hemos dicho, es el **=** y se encarga de guardar en la variable que figura a su izquierda el resultado de evaluar la expresión que figura a su derecha.

¿Cómo construimos y usamos los demás operadores de asignación, que llamamos combinados?

Debes recordarlo de la unidad 3, pero la forma general es algo parecido a lo que sigue:

Variable = variable operador expresión;

equivale a

variable operador = expresión

Lo mejor es verlo sobre el mismo ejemplo anterior, AritmeticanominasRevisada2.

En la línea 70 del código aparece la sentencia:

```
sueldoNeto = sueldoNeto + complemento;
```

Que puede trasformarse en :

```
sueldoNeto+=complemento;
```

Tú mismo puedes hacer el cambio y comprobar que todo sigue funcionando exactamente igual.

Unidad Didáctica VIII

Operadores de Incremento y Decremento

Incremento y Decremento expr++ , expr-- , ++expr , --expr

Frecuentemente nos podemos encontrar con situaciones en las que hay que contar las veces que ocurre algo. Por ejemplo, si quisiéramos contar la cantidad de trabajadores que hemos procesado con nuestra aplicación de nóminas, lo normal sería disponer de una variable `totalEmpleadosProcesados` de tipo entero que inicialmente valdría cero, y a la que se le sumaría uno cada vez que terminamos de procesar un nuevo empleado.

Podríamos escribir `totalEmpleadosProcesados= totalEmpleadosProcesados + 1;`

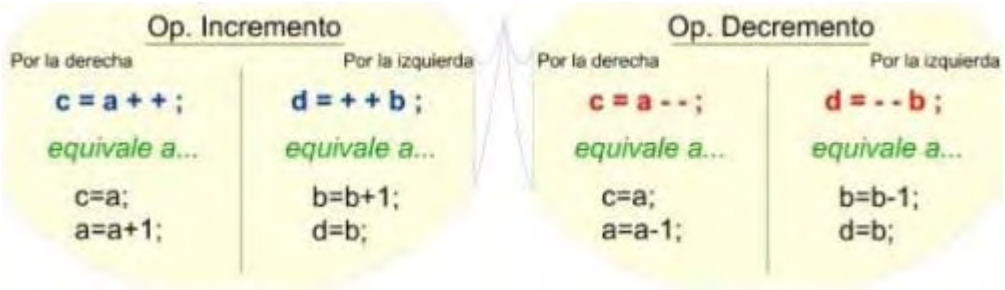
Pero la sentencia `totalEmpleadosProcesados++;` es más breve y hace exactamente lo mismo, de una manera igualmente clara.

También podríamos usar esa variable con el fin de contar los empleados que nos quedan por procesar. En este caso habría que asignarle inicialmente como valor el número de empleados a procesar, y restarle uno cada vez que terminamos de procesar otro empleado.

De nuevo podríamos escribir `totalEmpleadosProcesados= totalEmpleadosProcesados - 1;`

Pero la sentencia `totalEmpleadosProcesados--;` es más corta de escribir y hace exactamente lo mismo sin perder claridad.

Ambos operadores, de incremento y decremento, pueden usarse en notación prefija y postfija. La diferencia se aprecia cuando se usan en sentencias de asignación, es decir, colocando el operador a la izquierda o a la derecha de la variable que hace de operando.



Autoevaluación

Dados a =7 y b=8 señale la afirmación correcta:

- a) `int c = a++;` c valdrá 5.
- b) `int c = a++;` c valdrá 9.
- c) `int c = a++;` c valdrá 7.
- d) Ninguna de las anteriores es correcta.

Comprobar

Dados a =4 y b=8 señale la afirmación correcta:

- a) `int c = ++b;` c valdrá 9.
- b) `int c = ++b;` c valdrá 7.

- c) `int c = ++b;` c valdrá 10.
- d) Todas las respuestas anteriores son falsas.

Comprobar

Unidad Didáctica VIII

Operadores de bits

Operadores de Bits	Negación de bits	~
	Desplazamiento (bits)	<< , >> , >>>
	Conjunción bits (AND)	&
	Disy. excl. bits (XOR)	^
	Disy. incl. bits (OR)	|

¿Conocemos ya alguno de estos operadores? Realmente han aparecido todos en algún ejemplo visto hasta ahora, salvo los operadores de bits. Como ya dijimos en la unidad dedicada a los operadores, expresiones y sentencias, los operadores de bit podemos considerar casi sin miedo a equivocarnos que nunca vas a necesitar utilizarlos en las aplicaciones de gestión que aprenderás a desarrollar en este módulo profesional. Por tanto, no vamos a explicarlos aquí tampoco. Si por curiosidad queréis ver más detalles de su uso y funcionamiento, podéis ver el enlace **Para Saber Más** del punto dedicado a los operadores de bits de la unidad 3.

Unidad Didáctica VIII

Operador new de creación de objetos

Creación de objetos new

¿Recuerdas que en la unidad 2 se introducían los conceptos de clase y objeto? Como recordatorio, una clase viene a ser como una definición de un tipo de datos propios del usuario, y un objeto es cada uno de los valores que pueden tomar las variables del tipo que define esa clase. Aún no hemos entrado de lleno en los conceptos de clase y objeto , pero lo que sí puede que te hayas planteado es la necesidad de crear esos objetos, cómo crear objetos pertenecientes a una clase.

¿Hemos visto ya el operador de creación de objetos **new**? Realmente ha aparecido, pero de forma un poco secundaria, en el ejemplo de Gestión de Depósitos de la unidad anterior.

Habitualmente se utiliza este operador con la siguiente forma genérica:

<Nombre de la clase> <nombre de la variable objeto> **= new** <llamada al constructor de la misma clase>

Vamos a explicar su funcionamiento con el ejemplo en el que ha aparecido. En el ejemplo de Gestión de Depósitos, la clase GestionDepositos era la encargada de crear un par de depósitos y de operar con ellos. Las sentencias de creación de esos depósitos eran como las que siguen:

```
Deposito d1 = new Deposito("DEPOSITO1", "AGUA", 200);
Deposito d2 = new Deposito("DEPOSITO2", "ACEITE", 500, 250);
```

Realmente en ambos casos estamos haciendo prácticamente lo mismo. Ambas son sentencias de declaración e inicialización de una variable de tipo Depósito. Por ejemplo,

- **`Deposito d1`** está declarando que el identificador **`d1`** hace referencia a una variable de tipo **`Deposito`**, que es una clase definida aparte
- Luego le asignamos un valor con el operador de asignación **`=`**
- El valor asignado será el objeto depósito que devuelve la llamada al método **`Deposito ("DEPOSITO1","AGUA", 200)`**
- Este método está definido en la clase **`Deposito`** y al llamarse igual que la clase que lo contiene es interpretado por el compilador como el método que hay que usar para crear un objeto de ese tipo en la memoria.
- El operador **`new`** "ejecuta el constructor". Lo que hace es analizar la estructura de un objeto Depósito (un String para el nombre, un String para la sustancia a almacenar, un int para la capacidad y otro int para la cantidad contenida.
- Una vez analizada esa estructura, busca espacio libre en memoria donde poder alojar al nuevo objeto, y sigue las instrucciones indicadas por el método constructor para asignarle valores a los distintos campos del objeto.
- En concreto, tenemos dos constructores distintos ya que hemos sobrecargado el operador, y usamos uno con **`d1`** y otro con **`d2`**.
- En el caso de **`d1`**, el constructor crea un depósito de AGUA llamado DEPOSITO1 al que le caben 200 litros, y que inicialmente estará vacío.
- En el caso de **`d2`**, el constructor crea un depósito de ACEITE llamado DEPOSITO2 al que le caben 500 litros, y que se llena inicialmente con 250 litros de aceite.
- Finalmente se guarda en la referencia **`d1`** la dirección de memoria en la que ha creado ese objeto el operador new, que invocó al constructor, para que en adelante nos refiramos a él en el programa mediante ese identificador. Lo mismo ocurre con **`d2`**.

PARA SABER MÁS
Siguiendo con la página del manual de Java, podemos encontrar en este enlace detalles sobre el uso de objetos en Java. Te muestra ejemplos y nuevos enlaces para continuar profundizando en el tema.
Utilizando objetos Java. [Versión en caché]
Interesante enlace para que conozcas en que consiste la clonación de objetos en Java. Va a ser

muy útil cuando aprendas a utilizarla convenientemente.
Clonar objetos Java. [Versión en caché]

Autoevaluación

La forma habitual de usar el operador new tiene la siguiente forma genérica. Señala la afirmación correcta:

- a) <Nombre de la clase> <nombre de la variable objeto> **= new** <llamada al constructor de la misma clase>
- b) <Nombre de la variable objeto> <nombre de la clase> **= new** <llamada al constructor de la misma clase>
- c) **new** <Nombre de la clase> <nombre de la variable objeto> **=** <llamada al constructor de la misma clase>
- d) Ninguna de las anteriores es correcta.

Comprobar

Unidad Didáctica VIII

Operador de casting o conversión explícita de tipos.

Casting explícito (tipo)expr

El operador de casting explícito, consiste en poner un tipo entre paréntesis delante de alguna variable, constante, objeto, literal o expresión que sea de otro tipo. El resultado es que se fuerza la conversión del tipo de la variable al tipo indicado entre paréntesis, siempre que sea posible. Si no es posible, se producirá un error.

Un ejemplo de casting explícito lo hemos visto en esta misma unidad, en el apartado 2.4, en el ejemplo LiteralesEnterosErrorDeRepresentacion.java. En él, al sumar una variable de tipo byte consigo misma, obteníamos un valor fuera del rango del tipo byte. De hecho, obteníamos un valor de tipo int mayor que 127, que es el máximo valor para byte. Es por eso necesario forzar la conversión a byte del resultado de la suma para poder almacenarlo en una variable de tipo byte.

```
byte a = 125;
                a = (byte) (a + a);
```

Otro ejemplo de casting explícito lo podemos ver en el empleo de la función pow() de la clase Math para el cálculo de la potencia de dos números enteros. La función devuelve un valor double, ya que permite realizar potencias de números reales, pero nosotros necesitamos almacenar el valor en una variable de tipo int. Para ello hacemos un casting explícito como se ve a continuación.

```
int potencia =  (int) Math.pow(base,exponente);
```

Para ver el ejemplo completo, ejecuta el ejemplo contenido en el fichero Potencia.java, que aparece a continuación.

DEMO: Visualiza la ejecución del fichero Potencia.java

Descarga el archivo Potencia.java

Te recomendamos que también leas con atención las explicaciones que aparecen en los comentarios de ese ejemplo.

Autoevaluación

El resultado de usar el operador de casting explícito es: señala la afirmación correcta:

- a) El resultado es que se fuerza la conversión del tipo de la variable al tipo indicado entre paréntesis, siempre que sea posible. Si no es posible, se producirá un error.
- b) El resultado es que se fuerza la conversión del tipo de la variable al tipo indicado entre paréntesis siempre.
- c) La creación de nuevos objetos.
- d) Ninguna de las anteriores es correcta.

Comprobar

Unidad Didáctica VIII

Operador Condicional

Operador Condicional ? :

¿Para qué sirve el operador condicional?¿Ha aparecido ya en algún ejemplo en esta unidad?

Sí, ha aparecido en el ejemplo AritmeticaNominasRevisada2.java, necesitábamos preguntar si el trabajador estaba casado, respondiendo 0 si no está casado y 1 si sí lo está.

Después de preguntar, escribimos la sentencia

```
boolean casado = (respuesta == 1) ? true : false;
```

Esta sentencia comprueba lo que hemos respondido y asigna el valor adecuado a la variable lógica casado Usando el operador condicional **? :**

Es el único operador ternario de Java, es decir, el único que necesita tres operandos. Su forma general es:

<condición> ? <Expresión si condición verdadera> : <Expresión si condición falsa>

- El primer operando debe ser una expresión condicional, que se evalúa, y tomará el valor verdadero o falso.
- A continuación se pone el operador **?**
- Si el valor resultante de evaluar la condición es verdadero, el operador condicional devolverá el resultado de evaluar la primera expresión
- Si es falso devolverá el resultado de evaluar la segunda expresión.
- Ambas expresiones aparecen separadas por el carácter dos puntos (:)

El operador condicional no es más que una versión abreviada de una sentencia tipo if -then-else. De hecho, el mismo efecto podríamos haberlo conseguido usando una sentencia **if** de la siguiente forma:

```
bolean casado;
if (respuesta = = 1){
        casado = true;
}else {
        casado = false;
}
```

Un ejemplo de su uso puede ser para determinar el mayor de dos números enteros leídos desde teclado. Míralo en el ejemplo del siguiente enlace.

DEMO: Visualiza el programa que determina el mayor de dos números

Descarga el archivo Mayor.java

Autoevaluación

Respecto al operador condicional "? :" podemos afirmar que: señala la afirmación correcta:

a) El operador condicional no es más que una versión abreviada de una sentencia tipo if -then-else.
b) Es uno de los muchos operadores ternarios de Java.
c) Todas las anteriores son correctas.
d) Ninguna de las anteriores es correcta.

Comprobar

Unidad Didáctica VIII

Asociatividad y Precedencia de operadores.

Este apartado es un recordatorio de lo que ya expusimos en la unidad 3. Por eso nos limitamos a reproducir aquí la tabla en la que se indica el orden de precedencia de los operadores en Java, junto a la regla de asociatividad.

Recordemos también la finalidad de establecer un orden de precedencia de operadores y unas reglas de asociatividad: Poder escribir expresiones más cortas, sin necesitar usar tantos paréntesis, y de forma que quede perfectamente establecido el orden en que deben ejecutarse los distintos operadores que intervienen en una expresión.

No obstante, como ya se dijo, aprenderse el orden de precedencia de todos los operadores no suele ser necesario. Basta con saber que en la mayoría de los casos coincidirá con lo que estamos acostumbrados a asumir al escribir expresiones matemáticas. Y para los casos raros, cuando haya dudas, lo mejor es usar paréntesis.

Operadores postfijos	expr++, expr--
Operadores prefijos y unarios	++expr , --expr , +expr , -expr , ~ , !
Creación y cast	new , (tipo)expr
Multiplicativos	* , / , %
Aditivos	+ , -
Desplazamiento (bits)	<< , >> , >>>
Relacionales	< , <= , > , >=
Igualdad	= = , !=
Conjunción bits (AND)	&
Disy. excl. bits (XOR)	^
Disy. incl. bits (OR)	|
Conjuncion lógica	&&
Disyunción lógica	||
Condicional	? :
Asignación	+= , -= ,+= , *= , /= , %= , &= , |= , ^= , <<= , >>= , >>>=

Regla de asociatividad:
Todos los operadores son asociativos por la izquierda, es decir, a igual nivel de precedencia y a falta de paréntesis, se realizarán en el mismo orden que están escritos, excepto la asignación, que es asociativa por la derecha.

Puedes consultar los ejemplos de la unidad 3, si es que no tienes claro el significado de las reglas de asociatividad y precedencia de operadores.

Autoevaluación

Dada la siguiente expresión, k = l = m = 9, y teniendo en cuenta la regla de asociatividad, en Java se evaluará: señala la afirmación correcta:

- a) No sería evaluable, ya que se intentaría asignar a la variable k el valor de l, pero l todavía no tendría ningún valor asignado.
- b) En primer lugar se asigna 9 a la variable m, el cual se asigna a continuación a l, y posteriormente a la variable k
- c) En primer lugar se asigna 9 a la variable k, el cual se asigna a continuación a l, y posteriormente a la variable m.
- d) Todas las anteriores son falsas.

Comprobar

Señala la afirmación correcta. Respecto a la regla de asociatividad, podemos afirmar:

a) Todos los operadores son asociativos por la izquierda, es decir, a igual nivel de precedencia y a falta de paréntesis, se realizarán en el mismo orden que están escritos, excepto la asignación, que es asociativa por la derecha.

b) Resumiendo, podemos afirmar que en general todos los operadores son asociativos por la derecha.

c) Todos los operadores son asociativos por la derecha, excepto la asignación, que es asociativa por la izquierda

d) Todos los operadores son asociativos por la derecha, es decir, a igual nivel de precedencia y a falta de paréntesis, se realizarán en el mismo orden que están escritos, excepto la asignación, que es asociativa por la izquierda.

Comprobar

Dada la siguiente expresión, `k * k - 6 * m * p < 0`, indica como se analizaría (en Java):

a) `((k * k) - (6 * m * p)) < 0`.

b) `((k *k) - 6) * ( m * p)) < 0`.

c) `((k * (k - 6))* (m * p)) < 0`.

d) `k * k - 6 * m * p < 0`.

Comprobar