Programación

Bloque 01 - Introducción a la programación con Java

Índice

1.- Introducción

La Real Academia de la Lengua define la programación, entre otras definiciones, como "Elaboración de programas para su empleo en computadoras".

En este bloque nos iniciaremos en el mundo de la programación, específicamente en lenguaje Java, y aprenderemos los conceptos y la terminología básica necesarias para poder seguir el resto del curso y comprender la información que podamos obtener por otras vías, que suponen un conocimiento previo de estos conceptos.

2.- Bloques que componen un programa

Un programa está compuesto por uno o más bloques que juntos forman la estrutura del programa.

Los bloques varían de un lenguaje a otro, pero de forma clásica un programa ha estado compuesto por dos bloques:

- **Bloque de declaraciones.** Declara (de ahi su nombre) los datos que se van a utilizar en el programa, indicando la información necesaria sobre ellos para que el resto del programa los "conozca" y pueda decidir como tratarlos. Este bloque está formado por una o más:
  - **Declaración.** Una declaración informa al ordenador de que se va a necesitar espacio para un dato de un tamaño y características determinadas (que se definen mediante un tipo, como veremos más adelante).
- **Bloque de instrucciones.** Contiene las instrucciones del programa, propiamente dichas, junto con otras estructuras (repeticición, condiciones, etc.) que indican el orden en que se ejecutarán las mismas. Este bloque contiene una o mas:
  - **Instrucción.** Una instrucción es una orden que el ordenador comprende y sabe como realizar sin necesitar información adicional.

En los lenguajes orientados a objetos existe otra estructura adicional: La clase. Una clase es un bloque que engloba tanto declaraciones como instrucciones y es la unidad mínima en lenguajes orientados a objetos como Java. De hecho en Java TODO debe ir contenido dentro de una clase.

3.- Entornos Integrados de Desarrollo (IDEs)

Al igual que un diseñador gráfico necesita aplicaciones de diseño y un oficinista necesita aplicaciones ofimáticas, un programador también necesita aplicaciones que le faciliten la tarea a realizar.

Tradicionalmente un desarrollador utilizaba varias utilidades o programas, usualmente desde la línea de comandos, para realizar sus tareas. Poco a poco comenzaron a aparecer aplicaciones o paquetes de aplicaciones que servían para realizar TODAS las tareas relacionadas con el desarrollo de forma que el programador sólo tuviera que usar una única aplicación. A estas aplicaciones o paquetes se les denomina Entornos Integrados de Desarrollo (Integrated Development Environment o IDE).

En definitiva un IDE no es más que una aplicación o paquete de aplicaciones que integra todas las herramientas necesarias para el desarrollo y las conecta de forma que se puedan realizar las diferentes tareas que implica el desarrollo sin tener que andar cambiando de aplicación y proporcionando más productividad al desarrollo

En este curso vamos a aprender a programar usando el lenguaje Java, por lo que necesitamos un IDE que soporte este lenguaje. La oferta es amplia por lo que hemos elegido el IDE Eclipse.

Eclipse es un IDE gratuito y de código abierto, desarrollado por la Fundación Eclipse. Está basado en un antiguo producto de IBM (Websphere Studio). Se ha seleccionado por tener las siguientes ventajas:

- Gratuito. No cuesta dinero descargarlo e instalarlo. Asimismo sólo hay una versión a diferencia de otros IDEs que tienen una versión gratuita más o menos limitada y otra de pago con más funcionalidad. En Eclipse TODA la funcionalidad está disponible de forma gratuita
- Multiplataforma. Eclipse se puede instalar y usar en la totalidad de plataformas de escritorio más usadas hoy en día (Windows, Mac, Linux).
- Extensible. Eclipse dispone de una gran cantidad de extensiones o plugins que añaden funcionalidad a la disponible de forma inicial.
- Multilenguaje. Eclipse sirve para desarrollar en otros lenguajes, además de Java.

Obviamente, no todo van a ser ventajas y también tiene algunos inconvenientes:

- Algo pesado. Es un programa grande y tarda en cargar, sobre todo en equipos poco potentes. De todas formas un desarrollador debería disponer de un equipo potente puesto que el desarrollo es una tarea que requiere muchos recursos.
- Interfaz "feo". Algunos usuarios se quejan de que el interfaz es simple y poco atractivo. Para gustos los colores, aunque el autor reconocer que no le parece mal.
- Actualizaciones lentas. Los servidores de actualización no suelen tener un gran ancho de banda ni descargas rápidas por lo que las actualizaciones pueden ser algo lentas.

La instalación de Eclipse y del JDK se tratará aparte.

3.1.- Instalación de plugins para Eclipse

La instalación de plugins de Eclipse se tratará aparte.

4.- Proyectos en Eclipse

En Eclipse, como en la mayoría de IDEs, el trabajo se estructura en *proyectos*. Un proyecto es un conjunto de ficheros que juntos contribuyen para crear una aplicación. Esto incluye tantos archivos de programa (lo que se denomina *código fuente*) como otros archivos auxiliares necesarios para crear la aplicación (librerías, imágenes, textos, etc.). Como regla general, todos los archivos de un proyecto se almacenan en la misma carpeta en disco, aunque ésta se organiza internamente en subcarpetas, normalmente correspondientes a distintos tipos de ficheros.

Algunas de las operaciones que se realizan habitualmente con los proyectos son:

- Construir. Realiza las operaciones necesarias para generar el programa listo para ejecutar a partir de los ficheros que forman parte del proyecto (en inglés se lo conoce como Build o Render en proyectos más orientados al diseño)
- Limpiar. La construcción genera muchos archivos de trabajo o intermedios que se combinan para generar los archivos finales. Estos ficheros no se suelen eliminar después de construir porque en muchos casos se pueden reutilizar para otras construcciones, ahorrando tiempo. La eliminación o no de estos archivos se gestiona habitualmente de forma automática por el IDE pero a veces este automatismo falla y se hace necesario realizar una limpieza manual de forma que nos aseguremos que la siguiente construcción va a tomar en cuenta las versiones más recientes de todos los archivos (en inglés se conoce este comando como Clean o Clean All).

El proceso de creación y las tareas básicas con proyectos en Eclipse se tratarán aparte.

5.- Estructura mínima de un programa en Java

Tradicionalmente, cuando se comienza a aprender un lenguaje de programación, el primer programa que se realiza es el denominado "Hola Mundo". Este programa se llama así porque su objetivo es mostrar este mensaje (Hola Mundo) por pantalla al usuario y sirve para mostrar las capacidades básicas del lenguaje asi como el uso de las herramientas del lenguaje (o del IDE).

En Java, el programa Hola Mundo quedaría algo como lo siguiente:

```
1|public class HolaMundoApp {
2|  public static void main(String[] args) {
3|    // Imprimimos el mensaje
4|    System.out.println("¡Hola Mundo!");
5|  }
6|}
```

(Los números de línea se proporcionan por claridad pero no forman parte del programa ni hay que escribirlos. El código real es lo situado a la derecha de `|`)

La primera línea establece que vamos a crear una nueva clase (`class`), que es accesible desde cualquier otra clase (`public`) y que se llama `HolaMundoApp`. Finaliza la línea una llave de apertura (`{`) que, junto con su correspondiente llave de cierre (`}`), situada en la línea 6 marca el *contenido* de la clase.

Un detalle MUY importante a tener en cuenta es que Java distingue entre caracteres escritos en mayúscula o minúscula por lo que usualmente hay que escribir las palabras exactamente en formato de mayúsculas / minúsculas o se nos presentará un error por parte del sistema Java y el programa no compilará (ni funcionará). Por ejemplo, en la línea anterior la palabra `class` debe escribirse exactamente asi. Si se escribe con distintas mayúsculas y minúsculas (`Class` o `CLASS`) Java lanzará un error al intentar compilar el programa. Como regla general TODAS las *palabras clave* de Java, esto es, las palabras a las que Java da un significado especial van siempre completamente en minúsculas.

Este contenido de la clase, que es el contenido de las líneas 2 a 5, lo forma otra estructura, llamada *función* que ocupa estas líneas.

En la línea 2 se declara una función o *método* llamado `main`, que es pública, o sea que se puede acceder desde fuera de la clase (`public`), que está ligada a la clase, no a instancias particulares (`static`) y que no devuelve nada (`void`). La parte entre paréntesis se llama la `lista de parámetros`, que declara los posibles parámetros de entrada a la función (puede estar vacía si la función no admite parámetros). En nuestro caso declara un sólo parámetro (`String[] args`), que se llama `args` y su tipo es `String[]`.

La línea 3 contiene un texto precedido por dos barras (`//`). Esto es un *comentario*. Mas adelante hablaremos de los comentarios.

La línea 4 es la que contiene realmente la "chicha" de nuestro programa. Lo que hace es llamar a la función `println` del objeto `out` que es un miembro estático de la clase `System`. El significado exacto de este galimatías lo veremos en la siguiente unidad. Por ahora baste saber que esta instrucción lo que hace es mostrar por consola lo que se coloque entre los paréntesis. En este caso lo que hay es `"¡Hola Mundo!"`. Las comillas simples son la forma que tiene Java de permitir definir un *literal de cadena de caracteres*. Un *literal* es un valor que aparece en el código y que se debe interpretar tal y como está escrito. Una *cadena de caracteres* es una secuencia cualquiera de caracteres, incluida la secuencia vacia (sin ningún carácter). Un *caracter* es cualquier símbolo que el ordenador es capaz de manejar (letras, números, símbolos de puntuación, letras griegas, etc.). Por lo tanto `"¡Hola Mundo!"` es un literal (es un valor que se toma tal cual se escribe), de cadena de caracteres (de una secuencia de caracteres). Concretamente contiene la secuencia formada por el carácter `¡` seguido del carácter `H`, el carácter `o`, el carácter `l`, ... y por último el carácter `!`. Nótese que las comillas (`"`) no se tienen en cuenta. Su función es únicamente indicar donde comienza y acaba la secuencia pero no forman realmente parte de la misma.

Por lo tanto, esta secuencia es la que se imprime por consola y se mostrará en la salida, que es lo que queríamos que hiciera la aplicación.

Ya hemos terminado nuestra primera aplicación. Ahora hay que ponerla en marcha.

5.1.- Hola Mundo usando Eclipse

Vamos a ver el ciclo de trabajo clásico en Java utilizando Eclipse, que se tratará en documento aparte.

5.2.- Hola Mundo usando JDK

Vamos a ver el ciclo de trabajo clásico en Java utilizando las herramientas del JDK, que se tratará en documento aparte.

6.- Comentarios en Java

Como se ha comentado (valga la redundancia) anteriormente, un comentario en una construcción que tienen los lenguajes de programación que permiten insertar notas o mensajes en lenguaje natural dentro del código a fin de servir como documentación o recordatorio para los desarrolladores.

Dado que la nota o comentario no tiene sentido para la aplicación sino para los programadores, los sistemas descartan estos comentarios a la hora de construir o ejecutar los programas. Es como si no estuvieran ahi.

Todos los lenguajes soportan algún tipo de sintaxis para escribir comentarios. En Java existen dos tipos de comentarios: Comentarios simples y comentarios multilínea.

Un comentario simple se extiende desde la aparición de dos barras seguidas (//) hasta el final de la línea en la que aparece.

Ejemplo:

```
1|// Esto es un comentario
2|System.out.println("¡Hola Mundo!");  // Esto imprime el mensaje ¡Hola
Mundo!
```

En las dos líneas hay comentarios. La primera línea es todo un comentario con el texto `Esto es un comentario`

La segunda contiene una mezcla de código que se ejecuta (`System.out.println("¡Hola Mundo!");`) y comentario (`Esto imprime el mensaje ¡Hola Mundo!`).

Los comentarios multilínea pueden abarcar múltiples líneas (de hecho pueden abarcar todas las líneas que quieran, incluyendo 1). Un comentario multilínea comienza por la secuencia de caracteres /* y finaliza por la secuencia de caracteres */ Todo lo comprendido entre ambas secuencias se ignorará completamente a la hora de construir el programa.

Ejemplo:

```
1|/* Esto es un comentario */
2|/* Esto imprime
3|  el mensaje ¡Hola Mundo!
4|  por consola
5|*/
6|System.out.println("¡Hola Mundo!");
```

En este ejemplo hay dos comentarios. El primero está en la línea 1 y contiene el texto `Esto es un comentario`. El segundo se extiende por las líneas 2 a 5 y contiene el texto `Esto imprime el mensaje ¡Hola Mundo! por consola` (quitando los saltos de línea).

La línea 6 es una instrucción, no un comentario, y se ejecuta normalmente.

7.- Literales en Java

Java, como la mayoría de lenguajes, tiene las reglas necesarias para permitir expresar *literales*. Un literal es un valor fijo o constante que se introduce en un programa para ser utilizado en las instrucciones.

Esto es asi porque frecuentemente es necesario el utilizar valores constantes en cálculos o mensajes, como hemos visto en el ejemplo Hola Mundo. En aquel caso necesitabamos escribir exactamente el mensaje ¡Hola Mundo! y la única forma de indicar en el lenguaje que se usara ese valor exactamente es proporcionándoselo directamente. Esto es, usando un literal.

Java define reglas para varios tipos de literales. Esto es asi porque se permiten valores de distintos *tipos*.

7.1.- Tipos en Java

Java es un lenguaje considerado como *fuertemente tipado*. Esto significa que cualquier valor o dato que se maneja en un programa debe pertenecer a un *tipo*.

¿Qué es un tipo? Un *tipo de datos,* o *tipo* de forma abreviada, es un conjunto de posibles valores que puede tomar un dato de dicho tipo, junto con las operaciones permitidas y su semántica. Dicho en otras palabras, un tipo de datos `t` define:

- El conjunto de posibles valores que pertenecen a dicho tipo. Esto se puede especificar de forma exhaustiva, listando todos los posibles valores que puede tener un dato del tipo o mediante una regla que se aplica a un valor y determina si es un valor válido o no del tipo.
- Una serie de operaciones que se pueden realizar con datos de dicho tipo, junto con su significado, esto es, qué significa la operación y cómo se calcula, incluyendo casos límite o excepciones a las operaciones.

Ejemplo:

Java define, entre otros, un tipo llamado int (entero). Un dato de tipo int puede ser cualquier valor entero (sin decimales), con signo, comprendido entre -2147483648 y 2147483647, ambos inclusive. La razón de por qué estos extraños valores viene dada por el hecho de que un int se almacena en 32 bits, que permite 4294967296 valores. Como se necesitan valores negativos y positivos, la mitad de estos 4294967296 valores se usan como negativos y la otra como positivos (el 0 se toma como positivo, por eso el límite superior es 2147483647 en lugar de 2147483648)

Por lo tanto, dado un valor cualquiera se puede determinar si pertenece o no al tipo int usando un par de reglas:

- ¿Sigue la sintaxis de un número entero? Esto es, un signo (+ ó -, opcional en caso de que sea +) seguido de uno o más dígitos decimales.Si es asi, hay que consultar la siguiente regla. Si no lo es, no es un dato de tipo int
- ¿Está en el rango [-2147483648, 2147483647]? Si es así pertenece al tipo int, si está fuera NO pertenece el tipo int

Respecto a las operaciones, las siguientes estarían permitidas sobre datos de tipo int:

- Cambiar el signo. Se obtiene el entero con signo opuesto
- Sumar enteros. Se obtiene el entero resultante de sumar dos enteros
- Restar enteros. Se obtiene el entero (con signo) resultante de restar dos enteros.
- Producto de enteros. Se obtiene el producto de un entero por otro.
- División de enteros. Se obtiene el cociente de la división entera entre dos enteros (el resto se descarta).
- Módulo de enteros. Se obtiene el resto de dividir dos números enteros (el cociente se descarta).

7.2.- Tipos primitivos

Java proporciona los siguientes tipos primitivos (sólo veremos los valores. Las operaciones se describirán más adelante):

Tipo	Valores	Descripción
`boolean`	`true` ó `false`	Un valor lógico de tipo si/no, verdadero/falso.
`byte`	Números sin decimales con signo entre -128 y 127	Una cantidad entera (sin decimales y con signo) con menor o mayor rango de valores. Los de menor rango ocupan menos espacio en memoria que los de mayor rango. Por lo tanto se sacrifica número de valores por ahorro de espacio Excepto situaciones especiales nosotros usaremos siempre `int`
`short`	Números sin decimales con signo entre -32768 y32767	
`int`	Números sin decimales con signo entre -2147483648 y 2147483647	
`long`	Números sin decimales con signo entre $-9x10^{18}$ y $9x10^{18}$	
`float`	Números con decimales con signo entre $-3,4x10^{38}$ y $3,4x10^{38}$	Una cantidad real (con decimales y con signo) con menor o mayor rango de valores. Excepto situaciones especiales nosotros usaremos siempre `double`
`double`	Números con decimales con signo entre $-1,79x10^{300}$ y $1,79x10^{300}$	
`char`	Un carácter	Un sólo carácter. No confundir con una cadena de caracteres de un sólo carácter. Aunque puedan parecer lo mismo son datos distintos desde el punto de vista del lenguaje Java.

7.3.-

7.4.- Literales de los tipos primitivos

Los valores literales de los tipos primitivos se escriben según las reglas que se describen a continuación. Son bastantes porque Java permite una forma muy flexible de especificar literales, lo que complica algo las reglas aunque por contra facilita la vida al programador (cuando éste conoce las reglas, claro está)

7.4.1.- Literales de tipo booleano

El tipo booleano (`boolean`) tiene exactamente dos literales:

- `true`
- `false`

El valor `true` representa verdadero, si, afirmativo. El valor `false` representa falso, no, negativo.

7.4.2.- Literales de tipo entero

Los literales de tipo entero se pueden escribir usando distintas *bases de numeración*. Una base de numeración es el número que indica, para un sistema de numeración basado en la posición, como es el nuestro, que factor multiplicador vale más una posición respecto a la anterior. Por ejemplo, nosotros utilizamos (normalmente) un sistema de numeración de base 10. Esto es asi, porque cada dígito que aparece en un número vale, además del valor intrínseco del dígito, el valor de la posición que ocupa. Asi, en el número 6, el dígito 6 vale 6 porque ocupa la primera posición pero en 61, el

dígito 6 vale 60 (6 x 10) porque ocupa la segunda posición. En 632 el dígito 6 vale 600 (6 x 100) porque ocupa la tercera posición y así sucesivamente.

En informática, aparte de la común base 10 se suelen usar más sistemas de numeración, debido al hecho de que los ordenadores actuales utilizan internamente un sistema binario para almacenar los números. En un sistema binario (de base 2) se usan sólo 2 dígitos para representar los números y cada digito vale el doble que el situado en la posición anterior. El problema es que un número en base 2 necesita muchas cifras para representar una cantidad relativamente pequeña (por ejemplo, para representar el número 300 (en base 10) se necesitan 9 cifras(en base 2)). Por esto se suelen emplear bases alternativas como la base 8 ó la 16 que pueden representar cantidades con menos cifras pero son fácilmente convertibles a base 2 y viceversa (el método de conversión no lo veremos aqui). Un número en base 8 se representa usando los dígitos del 0 al 7 (8 dígitos distintos) mientras que un número en base 16 se representa usando los dígitos del 0 al 9 (10 dígitos distintos) y las letras de la A a la F para representar los 6 "dígitos" que faltan (El dígito A iría después del 9, el B después del A y asi suucesivamente).

En resumen, en Java se pueden escribir literales *enteros* en base 10 y además en base 2, 8 y 16.

7.4.2.1.- *Literales enteros en base 10*

Los literales enteros en base 10 pueden comenzar por un signo + ó - para indicar el signo (opcional en caso de +), seguidos de uno o más dígitos. Se pueden intercalar entre los dígitos el carácter _ para separar, al estilo de como se usa el punto al escribir números grandes para separar los miles, millones, etc.

Todos los literales enteros se toman como tipo `int`. Si se quiere un literal de tipo `long` hay que especificarlo explicitamente añadiendo la letra `l` ó `L` al final del número. Es un error intentar escribir un literal que esté fuera del rango de los `int` sin colocar la letra `l`.

Ejemplo de literales correctos	
Literal	**Comentario**
`10`	El número diez, positivo, `int`
`-10`	El número diez, negativo, `int`
`+10`	El número diez, positivo, `int`
`1_0`	El número diez, positivo, `int`
`-1_0`	El número diez, negativo, `int`
`-1_000_000`	El número un millon, negativo, `int`
`1_____0`	El número 10, positivo, `int`
`-10l`	El número diez, negativo, long
`10_0L`	El número 100, positivo, long
`3333333333l`	El número tres mil trescientos treinta y tres millones, trescientos treinta y tres mil trescientos treinta y tres, positivo, `long` (es obligatorio poner la `l` porque el rango excede el de `int`)
`-4_000_000_000L`	El número cuatro mil millones, negativo, `long` (la `L` es obligatoria al igual que en el ejemplo anterior).

Ejemplo de literales incorrectos	
Literal	**Comentario**
`1-0`	No es un literal sino la operación resta al valor 1 el valor 0
`_10`	El símbolo _ debe usarse para separar. No se puede colocar al inicio.
`1a`	a no es un dígito
`100 l`	La l está separada del número. Deben ir juntas.
`3333333333`	El número tres mil trescientos treinta y tres millones, trescientos treinta y tres mil trescientos treinta y tres, positivo. Es erróneo porque el rango excede el de `int` pero no lleva al final la `l`.
`-4_000_000_000`	El número cuatro mil millones, negativo, `long`. Es erróneo por la misma razón que el ejemplo anterior (falta la `l`)

7.4.2.2.- *Literales enteros en base 2*

Los literales enteros en base 2 comienzan por la combinación de caracteres `0b` ó `0B`, seguido de dígitos `0` ó `1`, en los que se puede intercalar, como en base 10, el carácter _ como separador. Las reglas para la determinación del tipo son las mismas que para los expresados en base 10, incluyendo el uso de la `l` para forzar a tipo `long`.

Ejemplo de literales correctos	
Literal	**Comentario**
`0b10`	El número dos, positivo, `int`
`0B10`	El número dos, positivo, `int`
`-0b10`	El número dos, negativo, `int`
`-0B10`	El número dos, negativo, `int`
`0b1_0`	El número dos, positivo, `int`
`-0B1_____0`	El número dos, negativo, `int`
`0b10000000000000000000000000000000l`	El número cuatro mil doscientos noventa y cuatro millones, novecientos sesenta y siete mil, doscientos noventa y seis, positivo, `long` (nótese la l minúscula al final)
`-0b10000000000000000000000000000000l`	El número cuatro mil doscientos noventa y cuatro millones, novecientos sesenta y siete mil, doscientos noventa y seis, negativo, `long` (nótese la l minúscula al final)
Ejemplo de literales incorrectos	
Literal	**Comentario**
`0b12`	Se usa un dígito o válido (2)
`0B_10`	El símbolo _ debe usarse para separar. No se puede colocar al inicio.

`0b1a`	a no es un dígito
`0 b100`	La `b` está separada del `0` inicial. Deben ir juntas.

7.4.2.3.- Literales enteros en base 8

Los literales enteros en base 8 (también llamada *octal*) están formados por el signo (con las mismas reglas que para base 10), un dígito `0` (es obligatorio que el primer dígito sea `0`), seguido de uno o más dígitos comprendidos entre `0` y `7` (ambos incluidos). Se aplican las mismas reglas ya vistas para los separadores y los valores `long`.

Ejemplo de literales correctos	
Literal	**Comentario**
`010`	El número ocho, positivo, `int`
`-010`	El número ocho, negativo, `int`
`+010`	El número ocho, positivo, `int`
`01_0`	El número ocho, positivo, `int`
`-01_0`	El número ocho, negativo, `int`
`-01_000_000`	El número doscientos sesenta y dos mil, ciento cuarenta y cuatro, negativo, `int`
`01_____0`	El número ocho, positivo, `int`
`-010l`	El número ocho, negativo, long
`010_0L`	El número sesenta y cuatro, positivo, long
`0123456701234l`	El número once mil, doscientos diecinueve millones, cuatrocientos sesenta y ocho mil, novecientos cincuenta y seis, positivo, `long` (es obligatorio poner la `l` porque el rango excede el de `int`)
`-076543210765L`	El número ocho mil, cuatrocientos catorce millones, seiscientos treinta mil, trescientos ochenta y nueve, negativo, `long` (la `L` es obligatoria al igual que en el ejemplo anterior).
Ejemplo de literales incorrectos	
Literal	**Comentario**
`078`	Contiene un dígito no permitido en base 8 (8)
`_010`	El símbolo _ debe usarse para separar. No se puede colocar al inicio.
`01a`	a no es un dígito
`0100 l`	La `l` está separada del número. Deben ir juntas.

7.4.2.4.- Literales enteros en base 16

Los literales enteros en base 16 (también llamada *hexadecimal*) están formados por el signo (con las mismas reglas que para base 10), la combinación de caracteres (sin espacios) `0x` ó `0X`, seguido de uno o más dígitos comprendidos entre `0` y `9` (ambos incluidos) o las letras entre la `a` y la `f`, ambas incluidas, mayúsculas o minúsculas. Se aplican las mismas reglas ya vistas para los separadores y los valores `long`.

Ejemplo de literales correctos	
Literal	**Comentario**
`0x10`	El número dieciseis, positivo, `int`
`-0X10`	El número dieciseis, negativo, `int`
`+0x10`	El número dieciseis, positivo, `int`
`0X1_0`	El número dieciseis, positivo, `int`
`-0x1_0`	El número dieciseis, negativo, `int`
`-0X1_000_000`	El número dieciséis millones, setecientos setenta y siete mil, doscientos dieciséis, negativo, `int`
`0x1_____0`	El número dieciseis, positivo, `int`
`-0X10l`	El número dieciseis, negativo, long
`0x10_0L`	El número doscientos cincuenta y seis, positivo, long
`0xabcdef012l`	El número cuarenta y seis mil, ciento dieciocho millones, cuatrocientos mil, dieciocho, positivo, `long` (es obligatorio poner la `l` porque el rango excede el de `int`)
`-0xAbC__dEF012L`	El número cuarenta y seis mil, ciento dieciocho millones, cuatrocientos mil, dieciocho, negativo, `long` (la `L` es obligatoria al igual que en el ejemplo anterior).
Ejemplo de literales incorrectos	
Literal	**Comentario**
`0x7G`	Contiene un dígito no permitido en base 16 (G)
`_0X10`	El símbolo _ debe usarse para separar. No se puede colocar al inicio.
`0x1ñ`	ñ no es un dígito en base 16
`0x100 l`	La `l` está separada del número. Deben ir juntas.

7.4.3.- Literales de tipo real

Los literales de tipo real se usan para expresar cantidades no enteras (con decimales).

A diferencia de los literales de tipo entero, los literales de tipo real siempre se escriben en base 10. Por defecto todos los literales reales se consideran de tipo `double`.

Hay dos formas de escribir literales de tipo real: La notación "normal" y la científica.

7.4.3.1.- Literales reales en notación normal

Los literales reales en notación normal se escriben con un signo `+` ó `-` (opcional en caso de `+`), seguido de uno o más dígitos, un punto (`.`) y más dígitos. El punto actúa como separador entre la parte entera (que queda a la izquierda del punto) y la parte decimal (que queda a la derecha). Sólo si la parte entera es `0` puede omitirse y el número comenzará por punto. Al igual que con los enteros se puede usar _ para espaciar o separar dígitos pero siempre debe ir **entre dígitos**.

Ejemplo de literales correctos

Literal	Comentario
`123.456`	El número ciento veintitres con cuatrocientos cincuenta y seis, positivo
`-234.567`	El número doscientos treinta y cuatro con quinientos sesenta y siete, positivo
`+345.678`	El número trescientos cuarenta y cinco con seiscientos setenta y ocho, positivo.
`.789`	El número cero con setecientos ochenta y nueve, positivo
`-.891`	El número cero con ochocientos noventa y uno, negativo
`+.901`	El número cero con novecientos uno, positivo
`4_561.1_789`	El número cuatro mil, quinientos sesenta y uno con mil, setecientos ochenta y nueve, positivo
Ejemplo de literales incorrectos	
Literal	**Comentario**
`*123.456`	El signo no es correcto (*)
`1a3.456`	Se ha usado un dígito incorrecto (a)
`123,457`	La separación entre la parte entera y decimal debe hacerse con punto
`123.`	Falta la parte decimal. No se puede omitir

7.4.3.2.- Literales reales en notación científica

Los literales reales en notación científica se escriben como los "normales", aplicando las mismas reglas y añadiendo al final del literal "normal" la letra `e` ó `E`, seguido de un número entero en base 10, con su signo si fuera necesario.

El valor del número se toma como la parte "normal" multiplicada por 10 elevada a la potencia expresada en el número entero indicado después de la `e`, al igual que se hace en matemáticas con la notación científica $2{,}3x10^4$, por ejemplo.

Ejemplo de literales correctos	
Literal	**Comentario**
`123.456e5`	El número doce millones, trescientos cuarenta y cinco mil, seiscientos (12345600), positivo
`-234.567E-2`	El número dos con treinta y cuatro mil, quinientos sesenta y siete (-2,34567), negativo
`+345.678E+2`	El número treinta y cuatro mil, quinientos sesenta y siete con ocho (34567,8), positivo.
`.789e-3`	El número cero con setecientas ochenta y nueve milmillonesimas (0,000789), positivo
`-.891E4`	El número ocho mil, novecientos diez (8910), negativo
`+.9_0_1e-10`	El número novecientas una billonésimas (0,0000000000901), positivo
Ejemplo de literales incorrectos	

Literal	Comentario
`*123.456`	El signo no es correcto (*)
`1a3.456`	Se ha usado un dígito incorrecto (`a`)
`123,457`	La separación entre la parte entera y decimal debe hacerse con punto
`123.`	Falta la parte decimal. No se puede omitir
`1.23 E-6`	Hay un espacio entre 3 y `E`
`1.45F6`	Para especificar la notación científica hay que usar `E`, no `F`.
`1.45E4.5`	El exponente (la parte a la derecha de `E`) debe ser entero, no real

7.4.4.- Literales de carácter

Antes de comenzar con los literales de carácter hay que introducir brevemente la forma en que los ordenadores tratan la información de tipo textual.

Los ordenadores, por su forma de funcionar, sólo pueden trabajar con números. Esto significa que cualquier dato que traten (texto, sonido, imágenes, pulsaciones de ratón o teclado, etc.) debe ser convertido a número para que el ordenador pueda tratarlo y posteriormente volver a convertirlo a su formato original cuando salga del ordenador.

En el caso de la información textual, los ordenadores utilizan una tabla de equivalencia, también llamada tabla de caracteres, juego de caracteres o codificación de caracteres, que establece una equivalencia entre números por un lado y los carácteres que representan cada uno. Por ejemplo, en una codificación el número 48 se corresponderá con el carácter a (a minúscula) mientras que en otro distinto puede que se corresponde con el carácter " (comillas dobles).

Debido a la forma en que ha evolucionado la informática, teniendo su origen en paises de habla inglesa y extendiéndose posteriormente al resto del mundo, las codificaciones de carácter han ido cambiando con el paso del tiempo, expandiendose con más caracteres conforme se han ido haciendo necesarios, lo que provoca que haya una gran cantidad de codificaciones y que muchas de ellas sean incompatibles entre si (porque el mismo carácter tiene códigos distintos en cada una de ellas).

Recientemente hay una codificación que se está imponiendo al resto. Se trata de la codificación llamada Unicode, creada y mantenida por el consorcio Unicode, cuyo propósito es crear una codificación universal que valga para todos los lenguajes humanos. Es una codificación que emplea números muy grandes porque hay idiomas (chino, por ejemplo) que tiene un caracter *por palabra*, con lo cual el número de caracteres para estos idiomas es bastante grande.

Java emplea Unicode como sistema de codificación para caracteres y cadenas de caracteres, por lo que a partir de ahora, cuando se hable de código ó código de carácter, se supondrá implícitamente que hablamos de código unicode.

Los literales de carácter (char) se expresan como un sólo carácter encerrado entre comillas simples ('). Por ejemplo, son literales de caracter `'a'` (la letra a minúscula), `'?'` (el cierre de interrogación) o `'{'` (llave de apertura).

Aunque este método es sencillo y rápido presenta un pequeño problema. De esta forma no es directo (a veces ni posible) el escribir literales de carácter para caracteres que no aparecen en el teclado que se está usando. Esto se puede paliar empleando aplicaciones del estilo del Mapa de

Caracteres de Windows pero puede ser problemático en otros equipos o configuraciones. Otro problema que se presenta es que no se puede escribir un literal para la comilla simple ya que si escribimos en Java `'''` Java lo toma como un literal incorrecto formado por las dos primeras comillas (incorrecto porque está vacío y debe contener un carácter) y una comilla suelta.

Para solucionar este problema se utiliza el denominado como *mecanismo de escapado (escaping)*. Este mecanismo consiste en dar un significado especial a un carácter determinado (en el caso de Java este caracter es \ (barra invertida)) que se empleará como caracter de escape. Esto significa que el carácter no vale su propio valor (en nuestro caso el carácter \ no se representa a si mismo) sino que sirve para indicar al sistema que lo que viene a continuación de él es una combinación especial destinada a representar un carácter de una forma alternativa.

Existen las siguientes combinaciones de escape:

Combinación	**Significado**
`'\\'`	Representa el carácter \ en sí
`'\b'`	Mover el cursor a la derecha
`'\t'`	Tabulador
`'\n'`	Salto de línea (pasar a la línea siguiente)
`'\r'`	Retorno de carro (volver el cursor al inicio de la línea)
`'\"'`	Comillas dobles
`'\''`	Comillas simples
`'\ooo'`	El número octal de tres cifras (deben ser exactamente 3) formado por los dígitos `ooo` indican el código Unicode del carácter a utilizar.
`'\uhhhh'`	El número hexadecimal de cuatro cifras (deben ser exactamente 4) formado por os dígitos `hhhh` indican el código Unicode del carácter a utilizar.

Ejemplos de caracteres por código:

- '\333' se corresponde con el carácter cuyo código unicode es 219 en decimal. Este código se corresponde en Unicode con el carácter Û (U mayúscula con acento circunflejo)
- '\u00Db' se corresponde con el mismo carácter que el anterior.

7.4.5.- Literales de cadena

Los literales de cadena son, salvando las distancias, muy parecidos a los literales de carácter. Las diferencias fundamentales son:

- Los literales de cadena se delimitan por comillas dobles (")
- Los literales de cadena pueden contener cero, uno o más de un carácter.
- Cada uno de los caracteres se introduce exactamente de la misma forma que los literales de carácter, esto es, el carácter en si se representa a si mismo y se usan las secuencias de escape para representar caracteres que no se encuentran en el teclado o no se pueden representar de otra manera.

Literal	**Valor real**

`"Hola"`	`Hola`
`"H\333la"`	`HÛla`
`"\""`	`"`
`""`	(vacía)
`"\'"`	`'`
`"Hol\u00DB"`	`HolÛ`

`"Hola Caracola"`	`Hola Caracola` (entre `a` y `C` hay un carácter que se llama espacio y que se usa para separar palabras en texto)
	No es lo mismo
	`Hola Caracola`
	que
	`HolaCaracola`
	La diferencia es este carácter espacio. Hay que tener en cuenta que forma parte de la cadena y se cuenta como contenido de la misma.

`" "`	(una cadena formada por un espacio en blanco)

8.- Variables

Un programa es una secuencia de instrucciones que realizan tareas sobre datos. Es decir, para hacer un programa necesitamos tanto un conjunto de datos sobre el que trabajar como las instrucciones que dicen qué hacer con esos datos.

Los literales que hemos visto anteriormente se usan en los programas cuando es necesario utilizar un dato con un valor fijo para ***todas*** las ejecuciones del programa, esto es, que todas las veces que se ejecute el programa se usará el mismo valor.

Esto no es el caso en la gran mayoría de las aplicaciones ya que se necesitarán tener datos almacenados en memoria que puedan almacenar distintos valores en cada ejecución del programa e incluso distintos valores en la misma ejecución del programa.

Pongamos un ejemplo, supongamos que queremos hacer un programa simple que nos permita calcular la media de dos números. El programa tendría poca utilidad si siempre calculara la media de los mismos números ya que el usuario lo único que obtendría es el mismo resultado siempre independientemente de lo que necesite.

Los programas utilizan dos tipos de dispositivos de almacenamiento para contener la información que se usa en un programa:

- La memoria principal. La memoria principal (conocida normalmente como memoria RAM) es la memoria de trabajo de la CPU. Es muy rápida de acceder y es la única que puede acceder la CPU directamente. Esto último significa que cualquier dato que queramos utilizar por la CPU debe estar obligatoriamente en memoria principal. Si no lo está, hay que copiarlo a memoria principal desde donde quiera que se encuentre. Un gran inconveniente de la memoria principal es que su contenido se pierde cuando finaliza el programa o se apaga el ordenador.
- La memoria secundaria. La memoria secundaria comprende cualquier otro tipo de dispositivo de almcenamiento distinto de la memoria principal. NO es accesible

directamente por la CPU por lo que si se quiere procesar un dato almacenado en memoria secundaria hay que copiarlo a la memoria principal y es más lenta (en muchos casos **mucho** más lenta) que la memoria principal. No todo son inconvenientes ya que la memoria secundaria mantiene normalmente su contenido aún cuando los programas no se están ejecutando o el ordenador está apagado, por lo que son útiles para el almacenamiento a largo plazo.

Por ahora nos vamos a centrar en el uso de la memoria principal (de ahora en adelante memoria a secas), dejando el uso de la memoria secundaria para más adelante en el curso.

Los literales que utilizamos en un programa se almacenan en memoria pero no se pueden modificar, teniendo un valor fijo cada vez que se ejecuta el programa. ¿Qué ocurre si necesitamos un dato que esté en memoria pero que pueda modificarse? Para este tipo de tareas Java proporciona el concepto de *variable*.

Una variable es un espacio en memoria (principal) que contiene un dato. El dato se puede tanto leer (acceder) como modificar (escribir). La lectura de una variable es *no destructiva*, esto es, el leer el valor de una variable no afecta para nada a su contenido. Por lo tanto podemos leer varias veces seguidas el contenido de una variable y siempre se leerá el mismo valor. Para variar el contenido de una variable se usa la operación de escritura, que modifica el valor almacenado en la misma y lo sustituye por uno nuevo. El valor existente anteriormente se pierde. En este sentido una variable sería como una pizarrita en la que sólo se puede esribir un dato. Se puede leer cuantas veces quieras pero para escribir hay que borrar lo existente y escribir encima.

En Java, para definir una variable hay que indicar tres cosas:

- Tipo de la variable. Esto le indica a Java que cantidad de espacio necesita reservar en la memoria para el dato y para comprobar que las instrucciones que utilizan el dato lo hacen siguiendo las reglas del tipo.
- Nombre de la variable. El nombre es una etiqueta que se asigna a la variable y que sirve para localizarla entre todas las posibles variables que se puedan tener en una aplicación. Dado que el nombre de una variable sirve para identificarla, el nombre debe ser único (veremos más cosas sobre los nombres más adelante)
- Valor inicial. Muchos lenguajes permiten definir una variable sin proporcionar un valor inicial. En este caso el valor inicial de la misma es indefinido y puede ser cualquiera. Java permite especificar un valor inicial opcional en el momento de crear la variable y este valor se usará cuando se cree la variable como primer valor de la misma. Si no se especifica no se podrá leer la variable hasta que no se le proporcione un valor mediante la instrucción de asignacioń. Esto impide que se usen variables cuyos valores no están definidos.
- Ámbito de la variable. El ámbito de una variable define el ciclo de vida de la misma, esto es, cuando se crea la variable, desde donde se puede acceder a ella y cuando se destruye. No hablaremos más de ámbito por ahora pero se tratará con más detalle en futuros temas.

8.1.- Nombres de variables

Los nombres de las variable los elige el programador de forma libre, siempre y cuando siga las siguientes reglas:

- No se pueden usar palabras reservadas del lenguaje. Palabras como `public`, `class`, `void`, `int`, etc. no se pueden usar como nombres de variable. Esto es asi para evitar confusiones.
- Un nombre de variable sólo puede contener letras, números y el carácter _. Aunque es posible usar caracteres propios del castellano como vocales acentuadas o la eñe, es altamente recomendable usar sólo letras que se usen en inglés para evitar problemas.
- Aunque son caracteres válidos, no se pueden usar números para el primer carácter, aunque si para el segundo y siguientes, esto es, `a1` es un nombre válido de variable pero `1a` no lo es.

Si cumplen las reglas anteriores, los nombre serán legales, pero además de legales, un buen programador elige nombres con significado.

Es muy importante elegir correctamente el nombre de una variable de forma que exprese de forma directa qué significa o qué uso tiene el dato contenido en la misma.

Por ejemplo, si necesitamos una variable para almacenar el resultado de sumar varios números es mucho mejor llamar a la variable `suma` que llamarla `s`, aunque ambos sean nombres legales. El primer nombre dice de un vistazo qué es lo que significa el dato que contiene la variable, mientras que el segundo no. Además, el poder dar (o no) nombre significativo a una variable da una idea al programador de que no tiene muy claro lo que está haciendo. Si tenemos un dato que empleamos en nuestro programa pero no tenemos claro como llamarlo, probablemente signifique que no hemos pensado correctamente qué datos necesitamos o qué hacer con ellos, lo que debe llevarnos a replantearnos el problema y nuestra solución.

8.2.- Creación de variables

Para declarar una variable en Java se usa la sintaxis:

```
tipo nombre=valor_inicial;
```

donde `tipo` especifica el tipo de la variable, `nombre` el nombre de la misma y la parte opcional `=valor_inicial` especifica el valor inicial. `valor_inicial` puede ser un literal del tipo adecuado (lo más usual) o el valor contenido en otra variable (ver más adelante para ver como se lee el valor contenido en una variable).

Esta instrucción tiene como resultado la creación de la variable con el nombre, tipo y valor inicial dados. A partir de ese momento (en las siguientes instrucciones) se podrá usar esa variable

8.3.- Uso de variables

Con las variables se pueden realizar dos operaciones:

- Acceder al contenido de la variable (leer).
- Modificar el contenido de la variable (escribir)

8.3.1.- Acceder al contenido de una variable

El acceder al contenido de una variable es muy sencillo. Simplemente usamos el nombre de la variable como si fuera un literal. Java lo interpreta como que se desea leer el contenido y "sustituye" el nombre de la variable por lo que vale

8.3.2.- Modificar el contenido de una variable

Para modificar el contenido de una variable se utiliza el operador = (igual) de la forma:

```
variable = valor;
```

Esta instrucción significa que se tome el valor `valor` y se escriba como contenido de la variable cuyo nombre es `variable`, eliminando el contenido anterior de la misma. Si la variable no se ha definido previamente se producirá un error.

Al operador = se le denomina *operador de asignación* porque *asigna* un valor a una variable. A la izquierda del operador sólo puede aparecer una variable ya definida previamente. A la derecha puede aparecer cualquier *expresión* que proporcione un valor del tipo de la variable. Ya veremos las expresiones más adelante. Por ahora vamos a usar dos tipos de expresiones:

- Literal. Un valor literal. Vale lo que vale el literal
- Variable. Una variable. Vale el dato contenido en la variable en cuestión.

Un aspecto importante a tener en cuenta es que el valor original que se asigna no se ve modificado de ninguna manera. Lo único que se modifica es el contenido de la variable a la que se asigna el valor, que cambia al nuevo valor asignado, perdiendo el valor existente.

Veamos el siguiente ejemplo:

```
1|public class Variables {
2|  public static void main(String[] args) {
3|    // Creamos una variable entera sin valor inicial
4|    int enteraSinValorInicial;
5|    // Otra entera pero con valor inicial (10)
6|    int enteraConValorInicial = 10;
7|    // Otra entera pero con valor inicial igual al valor de la anterior
8|    int enteraConValorInicial2 = enteraConValorInicial;
9|
10|    // Imprimimos un valor literal
11|    System.out.print("Un valor literal entero ");
12|    // El valor 2
13|    System.out.println(2);
14|    // Ahora imprimimos el contenido de la variable entera no
inicializada
15|    System.out.print("Valor contenido en la variable
enteraSinValorInicial ");
16|    // Lo que contiene la variable enteraSinValorInicial
17|    // Como no se inició no se puede usar. El siguiente comando comentado
no compilaría
18|    //System.out.println(enteraSinValorInicial);
19|
20|    // Ahora imprimimos el contenido de la variable entera inicializada
con literal
21|    System.out.print("Valor contenido en la variable
enteraConValorInicial ");
22|    // Lo que contiene la variable enteraConValorInicial
23|    // Esta si se puede usar porque está inicializada y su valor es 10
24|    System.out.println(enteraConValorInicial);
25|
26|    // Ahora imprimimos el contenido de la variable entera inicializada
con variable
27|    System.out.print("Valor contenido en la variable
enteraConValorInicial2 ");
```

```
28|    // Lo que contiene la variable enteraConValorInicial2
29|    // Como se inició con el contenido de la variable
enteraConValorInicial deberá valer lo mismo que ésta (10)
30|    System.out.println(enteraConValorInicial2);
31|
32|    // Ahora asignamos un valor a enteraSinValorInicial. Por lo tanto
ahora SI estará inicializada
33|    // Y se podrá usar
34|    enteraSinValorInicial = 20;
35|    // Al imprimirla debe salir 20
36|    System.out.print("Valor contenido en la variable
enteraSinValorInicial ");
37|    // Lo que contiene la variable enteraSinValorInicial
38|    System.out.println(enteraSinValorInicial);
39|
40|    // También podemos modificar una ya existente
41|    // enteraConValorInicial contiene actualmente el valor 10
42|    // Lo cambiamos a 100
43|    enteraConValorInicial = 100;
44|    // Al imprimirla debe salir 100
45|    System.out.print("Valor contenido en la variable
enteraConValorInicial ");
46|    // Lo que contiene la variable enteraConValorInicial
47|    System.out.println(enteraConValorInicial);
48|  }
49|}
```

Las líneas 1 y 2 son como las del ejemplo inicial pero ahora la clase se llama `Variable` en lugar de `HolaMundoApp`.

En la línea 4 creamos una nueva variable entera (`int`) sin inicializar llamada `enteraSinValorInicial`. Dado que esta variable no se ha inicializado no se permite leer su valor ya que el compilador lo detecta si se intenta hacer y produce un error.

En la línea 6 se crea otra nueva variable entera (`int`), llamada `enteraConValorInicial` pero esta vez si se inicializa al valor literal `10`.

En la línea 8 se crea la última variable entera (`int`), llamada `enteraConValorInicial2`. Esta también se inicializa pero usando el valor actual de la variable `enteraConValorInicial`. Como en este punto de la ejecución la variable `enteraConValorInicial` contiene el valor `10`, este es el valor con el que se inicia también la variable `enteraConValorInicial2`.

En la línea 11 se imprime un texto fijo. Hay que notar que, a diferencia de lo usado anteriormente, aqui se usa la función `System.out.print` en lugar de `System.out.println`. Las dos hacen prácticamente lo mismo (imprimir algo por consola) con la diferencia de que `println` genera un salto de línea y retorno de carro después de imprimir lo que se le proporcione mientras que `print` se limita a imprimir lo que se le proporciona. Si se vuelve a hacer otro `print` a continuación, el texto aparecerá en la consola justo detrás de lo que se acaba de imprimir. Esto se puede usar para combinar distintos tipos de salida en una sola línea, aunque hay otros métodos mejores y más flexibles que veremos más adelante.

En la línea 13 se imprime el valor literal entero 2, que es exactamente lo que sale por pantalla.

En las líneas 14 a 18 se intentaría imprimir, al mismo estilo que las líneas previas, una pequeña cabecera o texto fijo y el valor de la variable `enteraSinValorInicial`. El problema es que la

instrucción que se muestra en el comentario de la línea 18 no se puede dejar sin comentar porque impide que el programa compile. Esto es asi porque la variable `enteraSinValorInicial` no se ha inicializado y por tanto al intentar acceder a su contenido el compilador detecta la circunstancia anómala y detiene la compilación, por lo que el programa no funcionará.

En las líneas 20 a 24 se intenta imprimir el contenido de la variable `enteraConValorInicial`. En este caso no hay problema y se debería imprimir el valor `10`, que es el que se proporcionó en la inicialización de dicha variable en la línea 6.

En las líneas 26 a 30 se intenta imprimir el contenido de la variable `enteraConValorInicial2`. En este caso tampoco hay problema y se debería imprimir el valor `10`, que es que se le proporcionó en la inicialización de la línea 8 (también `10`, como la variable `enteraConValorInicial`).

En la línea 34 asignamos el valor literal `20` a la variable `enteraSinValorInicial`. Esto tiene dos efectos: Por un lado la variable `enteraSinValorInicial` pasa de estar NO inicializada a estar inicializada. Por otro se almacena en la misma el valor de la derecha de la asignación. En este caso el valor se ha proporcionado mediante un literal (`20`), por lo que este será el nuevo valor para el contenido de la variable.

En las líneas 35 a 38 se intenta imprimir de nuevo el contenido de la variable `enteraSinValorInicial`. Ahora esta instrucción no provoca error porque la variable ya está inicializada por lo que funciona correctamente e imprime su valor actual (`20`).

En la linea 43 asignamos el valor `100` a la variable `enteraConValorInicial`. Esta variable contenía antes de esta instrucción el valor `10` y después contendrá el valor `100`. El valor anterior (`10`) se pierde.

En las líneas 44 a 47 se imprime el valor de la variable `enteraConValorInicial`. Como su valor ahora es `100`, ya que se modificó en la línea 43, ahora debería imprimirse `100`.

En este ejemplo, un poco largo, hemos visto como se crean, usan y modifican variables. Por supuesto el programa no tiene un propósito útil más allá de demostrar estas cosas. Cuando se introduzcan las expresiones y la lectura de datos veremos como hacer programas que tengan alguna utilidad práctica.

9.- Constantes

Hasta ahora hemos usado de forma libre literales de diversos tipos en nuestros programas. Esto está bien pero se puede mejorar. Cuando hagamos programas más largos veremos que aparecen salpicados aqui y allá literales a lo largo del mismo. Muchos de ellos son de un solo uso (por ejemplo la mayoría de los literales de cadena que estamos usando para imprimir mensajes) pero hay otros que tienen un significado concreto y que además se podrían usar más de una vez en un programa. Un ejemplo es el valor π (pi) que equivale apróximadamente a 3,1415926. Este valor se puede usar en un programa para realizar varios cálculos, por ejemplo, la longitud de la circunferencia y el área de un círculo a partir de la longitud de su radio. En este caso habría que utilizarlo un par de veces pero un número simplemente no indica qué significa aunque en este caso sea un número famoso y fácilmente reconocible. En otros casos no es tan obvio qué es ese número

ni para que sirve cuando se lee el programa por parte de otro programador o por el mismo después de pasar un tiempo.

Para solucionar el problema vamos a introducir el concepto de *constante*. Una constante es un valor al que le damos un nombre, igual que una variable pero que, a diferencia de las variables, no se puede modificar una vez se le ha dado un valor inicial.

Las constantes se definen de una forma similar a las variables pero con alguna diferencia. En primer lugar la declaración se coloca entre la declaración de la clase (`class` ....) y la de la función main (`public static void main` .....). Si hay más de una se colocan todas en esa posición, cada una en su línea. En segundo lugar la forma de la declaración es ligeramente distinta a la de una variable. Hay que usar la sintaxis:

```
public static final tipo nombre = valor_de_la_constante;
```

donde:

- `public static final` es la forma de indicar que lo que se está declarando es una constante y no una variable. En posteriores unidades ahondaremos en el significado de estas esotéricas palabras. Por ahora baste saber que las tres son reservadas y que hay que usarlas en este orden al inicio de la definición de una constante.
- `tipo` es el tipo de la constante (`int`, `long`, `double`, etc.)
- `nombre` es el nombre de la constante. El nombre debe seguir las mismas reglas que el una variable. Sin embargo, por convención, para las constantes se usan nombres con todas las letras en mayúsculas y se usa el símbolo _ para separar distintas palabras que pudieran aparecer en el mismo. Por ejemplo para definir una constante que indica el número de usuarios que se van a gestionar se podría usar el nombre `NUMERO_USUARIOS`
- `= valor_de_la_constante` sirve para indicar el valor que va a contener la constante. Usualmente hay que usar un literal en el lugar de `valor_de_la_constante`

Ejemplos:

```
public static final double PI = 3.1415926;
```

declara la constante real llamada `PI` con un valor apróximado de la misma

```
public static final int LONGITUD_MAXIMA = 5;
```

define una constante entera llamada `LONGITUD_MAXIMA` que vale `5`.

Una ventaja añadida de las constantes, además de "dar un nombre a un valor" es la de que si el valor cambia sólo hay que hacerlo en un sitio del programa. Si hemos copiado y pegado el literal a lo largo del programa y el valor cambia hay que buscar y reemplazar todas las apariciones del literal correspondiente a lo largo del código. Peor aún, si hay más de una constante con el mismo valor hay que ir examinando cada aparición del literal y determinar si hay que cambiarlo o no.

10.- Análisis de datos y tipos

Cuando vayamos a realizar cualquier programa, el primer paso a realizar, antes de escribir una sola instrucción, es analizar el problema y determinar qué datos necesitamos para poder solucionarlo con un programa.

Los datos aparecen habitualmente como *nombres* en el enunciado o descripción del problema. Por lo tanto, examinando los nombres que aparecen en el enunciado se puede determinar buena parte de los datos involucrados en el proceso que se desea realizar.

El segundo paso es determinar la forma en que se emplea el dato en el programa. Básicamente hay tres formas de emplear un dato en un programa:

- **Dato de entrada.** Es un dato que se proporciona a la aplicación para que realice su tarea. Por ejemplo, si tenemos un programa que calcula la media de tres números, estos tres números son datos de entrada a la aplicación ya que son los datos de los que se alimenta el proceso para poder realizarse.
- **Dato de salida.** Es un dato que la aplicación genera o calcula y forma parte del resultado del proceso. Por ejemplo, en el ejemplo anterior, la media es un dato de salida, ya que es calculado por el proceso y es el producto del mismo.
- **Dato de maniobra o temporal.** Es un dato que se emplea para una parte intermedia del cálculo. Por ejemplo, para el ejemplo anterior, la suma de los tres números es un dato de maniobra ya que se necesita como paso intermedio para el cálculo del resultado (la media) pero una vez calculada esta ya no se necesita, de ahi también el nombre de temporal (sólo se necesita durante un tiempo).

El siguiente paso es determinar, para cada dato, cual es su *tipo*. Esto es importante porque el tipo de un dato influye tanto en los posibles valores que podría contener como en las operaciones que se pueden realizar sobre él. Una regla general para determinar el tipo de un dato es la siguiente:

1. ¿Es el dato un valor de verdad / falsedad (o si / no)? Si es el caso estamos ante un dato de tipo boolean.
2. Si la respuesta a la anterior pregunta es no, ¿El dato es una *cantidad*, esto es, tendría sentido realizar operaciones aritméticas con el dato (por ejemplo sumarlo con otro dato)? Si la respuesta es no, pasar a la pregunta 5.
3. Si la respuesta a 2 es si, ¿vamos a necesitar que el dato pueda tener decimales? Hay que pensar no solo en los valores más usuales que puede tener el dato sino en valores no usuales o extremos. Si la respuesta es si (o podría), usar el tipo `double`, si es no (nunca) pasar a la pregunta siguiente.
4. ¿El dato puede contener valores muy grandes (en el orden de los miles de millones)? Si es que si, usar el tipo `long`. Si es no, usar el tipo `int`. Los tipos byte y short tienen usos muy especializados que se tratarán en su momento. Como norma general sólo vamos a usar `int` y `long` en este curso para datos sin decimales o enteros.
5. Si el dato NO es una cantidad, entonces es tipo textual. ¿El texto va a consistir SIEMPRE de un sólo carácter? Si la respuesta es si, el dato debe ser de tipo `char`. Si la respuesta es no, incluyedo el caso de que el dato pueda en alguna ocasión no contener ningún carácter, el dato es una cadena de caracteres.

Por último nos queda determinar si el dato será un literal, constante o variable. La regla general es la siguiente:

- Si el dato puede cambiar durante la ejecución del programa o entre distintas ejecuciones del mismo programa, se debe usar una variable.
- Si el dato no cambia ni durante la ejecución del programa ni entre distintas ejecuciones del mismo programa, se puede usar tanto un literal como una constante. ¿Cuando usar una u otra?
- Si el dato que no cambia no tiene significado propio, usaremos un literal. Una pista fuerte sobre si el dato tiene o no significado propio es si nos cuesta ponerle un nombre o el nombre es más o menos el mismo que el valor. Por ejemplo, para calcular la energía cinética (e) de un móvil a partir de su masa (m) y su velocidad (v) se emplea la fórmula e = (m * v * v) / 2. En este caso tanto e como m como v serían (o podrían ser variables) pero el 2.... ¿Podríamos ponerle nombre? A mi no se me ocurre otro que DOS o algo similar. Por lo tanto lo dejamos como literal (2).
- Si el dato SI tiene significado propio, usaremos una constante. La pista es la misma que en el paso anterior (es relativamente fácil asignar un nombre con sentido). Por ejemplo, para calcular la longitud de una circunferencia (l) a partir del radio (r) se puede usar la fórmula: l = 2 * 3,1415926 * r. En este caso 2 estaría en el mismo caso que el paso anterior pero a 3,1415926 si es fácil asignarle un nombre: PI. Por lo tanto sería una constante.
- Otra pista para diferenciar entre constante y literal es por el uso, aunque no es tan fuerte como la anterior y puede llevar a constantes "raras". Por ejemplo, si en un programa usáramos las dos fórmulas anteriores, en ambas aparece el número 2. Ya que se usa "frecuentemente" (si a dos apariciones se le puede llamar asi) podríamos usar una constante (por ejemplo con el horrible nombre de DOS) y usarla en las dos fórmulas. El problema es que el valor 2 en una y otra fórmula son independientes y uno podría cambiar sin que se necesite cambiar en la otra fórmula. Por lo tanto no son candidatos a ser constantes y es mejor dejarlos como literales.

11.- Introducción de datos por el usuario

A fin de que las aplicaciones que realicemos sean más útiles vamos a introducir un mecanismo que nos permite obtener datos introducidos por el usuario. De esta forma las aplicaciones serán más flexibles puesto que los datos sobre los que se va a operar los decide el usuario en el momento de usar la aplicación, en lugar de estar colocados directamente en el programa.

Los datos los introducirá el usuario mediante teclado, indicando la finalización de la entrada pulsando la tecla Intro (o Enter).

Para poder hacer esto se os pide un acto de fe y que se use lo que se va a describir a continuación sin explicar como funciona. Lo importante es que funciona y hace lo que se necesita. Más adelante en el curso se despejarán dudas sobre el funcionamiento de lo que ahora vamos a ver.

El siguiente ejemplo lee (y escribe) usando la consola datos en diversos tipos (vamos a eliminar los comentarios para ahorrar espacio):

```
1 import java.util.Scanner;
2
3 public class LecturaTecladoApp {
4   public static void main(String[] args) {
```

```
5|    Scanner sc = new Scanner(System.in);
6|
7|    System.out.print("Introduce un valor entero: ");
8|    int valorEntero = Integer.parseInt(sc.nextLine());
9|
10|    System.out.print("Introduce un valor entero largo: ");
11|    long valorEnteroLargo = Long.parseLong(sc.nextLine());
12|
13|    System.out.print("Introduce un valor real: ");
14|    double valorReal = Double.parseDouble(sc.nextLine());
15|
16|    System.out.print("Introduce un valor de carácter (un sólo carácter):
");
17|    char valorCaracter = sc.nextLine().charAt(0);
18|
19|    System.out.println("El entero introducido es " + valorEntero);
20|    System.out.println("El entero largo introducido es " +
valorEnteroLargo);
21|    System.out.println("El real introducido es " + valorReal);
22|    System.out.println("El carácter introducido es " + valorCaracter);
23|  }
24|}
```

En la línea 1 se utiliza la instrucción `import` (que es una palabra reservada) para indicar que vamos a usar la clase `Scanner` (el significado de esto lo veremos en próximas unidades. Por ahora se copia y pega esta línea tal cual al inicio del fichero siempre que se quiera usar la clase `Scanner`).

En las líneas 3 y 4 se declara la clase de este programa, `LecturaTecladoApp`, y la función `main`, como ya viene siendo habitual.

En la línea 5 definimos e inicializamos un objeto de la clase `Scanner` (`sc`). Esta línea hay que copiarla tal cual al inicio de la función main de cualquier programa que hagamos que necesite leer desde teclado. A partir de ese momento se usará la variable `sc` para realizar las lecturas desde teclado.

En las líneas 7 y 8 se solicita un dato entero al usuario para que lo introduzca por teclado. La lectura realmente se realiza en la línea 8. La instrucción de la línea 7 sirve para mostrar por pantalla un *prompt* al usuario. Un prompt es un texto informativo que se muestra al usuario para indicarle que debe introducir un dato o realizar una acción. Se podría realizar la lectura sin necesidad de prompt pero asi los programas son más amigables ya que se informa y guía al usuario sobre lo que se espera de él en cada momento. Si se presenta un cursor pidiendo un dato se puede confundir al usuario que puede no saber qué es el dato que se pide, de que tipo es ni para qué se va a emplear. Usando un prompt ayudamos al usuario a tomar una decisión informada y a elegir un dato correcto y adecuado a lo que se pretende hacer. Vamos a desgranar un poco lo que se hace en la línea 8:

```
int valorEntero = Integer.parseInt(sc.nextLine());
```

Al inicio (`int valorEntero`) se está definiendo una variable llamada valorEntero de tipo int. Al mismo tiempo se está inicializando (`=`) y el valor que se toma se obtiene de la expresión `Integer.parseInt(sc.nextLine())`. Esta expresión se debe utilizar siempre que se quiera obtener un dato de tipo entero que introduzca el usuario desde teclado, tal cual está escrita y sin modificaciones. Cuando el usuario termine de introducir el dato y pulse la tecla Intro, se genera (de forma mágica por ahora) un valor de tipo entero que se corresponde con lo que el usuario ha

tecleado. Las reglas que debe cumplir el número introducido son las mismas que para los literales de tipo entero.

En la línea 11 se lee un entero largo. En este caso la expresión mágica para leer el valor desde teclado es `Long.parseLong(sc.nextLine())`.

En la línea 14 se lee un valor real. En este caso la expresión mágica para leer el valor es `Double.parseDouble(sc.nextLine())`. Cuidado porque para separar la parte entera de la decimal se usa el punto (`.`) no la coma (`,`) como se suele hacer en España.

En la línea 17 se lee un carácter. La expresión es `sc.nextLine().charAt(0)`.

Una precaución que hay que tener en cuenta es que si el valor que se introduce no cumple las reglas del tipo en cuestión se produce un error y el programa termina inmediatamente.

Por último en las líneas 19 a 22 se imprimen por pantalla los valores anteriormente leidos, que deberían coincidir. Hay que resaltar la forma en que se realiza la impresión. Como se ve puede ver, el valor que se imprime consta de un literal de cadena seguido del símbolo `+` y la variable con el valor a imprimir. En este caso el símbolo + actúa como símbolo de unión de cadenas. Lo que hace es formar una cadena con los contenidos de la primera (el literal) más el resultado de convertir a cadena el segundo (el valor numérico o de carácter contenido en la variable). Veremos más sobre esto en la siguiente unidad. Por ahora baste saber que se puede usar y debe funcionar.

12.- Expresiones

Cuando hablamos de lenguajes de programación en general, y de Java en particular, una *expresion* es una combinación de valores literales, variables, constantes y operadores (también funciones, que veremos en unidades posteriores) que es interpretada y evaluada de acuerdo con las reglas del lenguaje para obtener un resultado. Se podría ver como una instrucción que indica al ordenador que realice un cálculo y obtenga un resultado. La diferencia con una instrucción es que las expresiones no llevan una palabra u orden explícita. Cuando el sistema o el compilador encuentra una expresión la traduce a las instrucciones adecuadas para calcularla. El resultado final de una expresión es un valor, del tipo correspondiente a la expresión (más sobre esto más adelante) que *sustituye* a la expresión.

Por ejemplo, si en Java nos encontramos la expresión:

```
2 + 3
```

Java la reconoce como una expresión formada por los literales `2` y `3` y el operador `+`. Dado que la expresión es correcta en Java, éste la traduce por el cálculo de la suma (dado que el operador es `+`) de los dos valores situados a izquierda (`2`) y derecha (`3`). Este cálculo produce el resultado `5`, que sustituye a la expresión, siendo el resultado neto como si hubiéramos escrito directamente en el programa:

```
5
```

Las expresiones se pueden hacer tan complejas como se necesiten combinando distintos operadores y valores, siempre y cuando se respeten las reglas que se verán a continuación y que los tipos de los valores sean compatibles con las operaciones que se van a realizar con ellos.

Por ahora vamos a ver los siguientes operadores:

- Ariméticos. Realizan las operaciones aritméticas comunes sobre valores numéricos (suma, resta, producto, etc.) y devuelven a su vez *un* valor numérico.
- De comparación. Comparan dos valores y devuelve un valor de tipo booleano (true / false).
- Lógicos. Toman uno o dos valores booleanos y devuelven otro valor booleano.

Hay que tener en cuenta que las expresiones, tal y como las vamos a ver en esta unidad, no afectan para nada a los valores de las posibles variables que puedan participar en las mismas y que mantendrán su valor al terminar el cálculo.

Otro detalle a tener en cuenta es que, independientemente de los operadores usados, también se pueden emplear paréntesis para agrupar partes de la expresión, ya sea para que se calculen antes o para dejar claro si participan en un cálculo o no.

12.1.- Operadores aritméticos

Los operadores aritméticos se pueden emplear con datos de tipo numérico (entero o real). No se pueden emplear con datos de otros tipos (o por lo menos no con el significado que aqui se describe. Ya veremos el uso de algunos de los operadores que describimos aqui sobre otros tipos da datos más adelante ya que realmente no tienen el mismo significado que se describe en esta sección)

El resultado de la operación será entera si los tipos de **todos** los operandos son enteros. Si alguno es real, el resultado será real. Por ejemplo `2 + 3 = 5`, (tanto `2`, como `3` como `5` son enteros) mientras que `2 + 3.0 = 5.0` (`2` es entero y `3.0` es real, por lo que el resultado también lo será (`5.0`))

Existen ocho operadores aritméticos:

- Cambio de signo (`-`). El cambio de signo cambia el signo del valor al que precede. Es un operador unario (sólo tiene un operando). Ejemplo: `--2` vale `2` (ó `+2`), `--3.4` vale `3.4`.
- Suma (`+`). La suma toma dos operandos y los sustituye por el resultado de sumarlos. Ejemplo: `2 + 3` vale `5`, `2 + -3` vale `-1`, `3.4 + 2.7` vale `6.1`, `2 + 3.5` vale `5.5`
- Resta o diferencia (`-`). Este operador, cuando se usa entre dos operandos toma ambos y resta el segundo del primero y los sustituye por el resultado. Ejemplo: `2 - 3` vale `-1`, `2 - -3` vale `5`, `3.4 - 2.7` vale `0.7`, `2 - 3.5` vale `-1.5`
- Producto o mutiplicación (`*`). Este operador toma dos operandos y los sustituye por el producto de ambos. Ejemplo: `2 * 3` vale `6`, `2 * -3` vale `-6, 3.4 * 2.7` vale `9.18`, `2 * 3.5` vale `7.0`.
- División (`/`). Este operador tiene un comportamiento distintos según los tipos de los operandos:
  - Operandos enteros. Si **ambos** operandos son enteros, el resultado es el cociente de la división entera del primero entre el segundo. Ejemplo: `3 / 2` vale `1`, `8 / 5` vale `1`
  - Operandos reales. Si **algún** operando es real (o ambos), el resultado es la división real del primero entre el segundo. Ejemplo `3.0 / 2` vale `1.5`, `8.0 / 5.0` vale `1.6`.

Un apunte importante. Si el segundo operando (el divisor) vale 0, la operación no se puede realizar (no está definida) y se producirá un error que detendrá el programa.

- Módulo (%). Este operador sólo se puede usar si **ambos** operandos son enteros. No aplica si alguno o los dos son reales. EL operador calcula la diivisión entera del primer operando entre el segundo y devuelve el *resto* de la operación. El cociente se descarta. Ejemplo `3 % 2` vale `1` (`3` entre `2` da como cociente `1` y resto `1`), `8 % 5` vale `3` (`8` entre `5` da como cociente `1` y resto `3`).
- Incremento (`++`). Este operador sólo toma un operando, **que debe ser obligatoriamente una variable entera (`int` o `long`)**. El operador toma el valor actual de la variable, le suma uno y coloca el valor actualizado en la misma variable. El operador se puede colocar antes o después de la variable y esto afecta al valor que devuelve el expresión. Por ejemplo, suponiendo que la variable a vale 10, si imprimimos ++a se imprime 10 (aunque la variable a pase a valer 11 tras la operación, se toma el valor que tiene antes de hacerla), pero si imprimimos a++ se imprimirá el valor 11 (se toma el valor de la variable después de hacer la operación).
- Decremento (`--`). Este operador es el opuesto al anterior. En lugar de incrementar en uno decrementa el uno pero el resto de la descripción se le aplica igualmente.

Una cuestión que puede surgir es el orden en que se ejecutan los cálculos cuando una expresión consta de más de un operador. En algunos casos este orden es indiferente respecto al resultado final y en otros puede ocasionar que se calculen resultados distintos. Por ejemplo, la expresión `2 + 3 + 4` vale `9` de resultado final sea cual sea el orden en que se realicen los cálculos (`2 + 3 + 4 = 5 + 4 = 9` ó `2 + 3 + 4 = 2 + 7 = 9`). Pero, en cambio, la expresión `2 + 3 * 5` da resultados distintos según el orden que se siga a la hora de hacer los cálculos (`2 + 3 * 5 = 5 * 5 = 25` ó `2 + 3 * 5 = 2 + 15 = 17`)

El orden concreto en el cual se ejecutan los cálculos sigue una regla llamada de *precedencia de operadores*. Esta regla establece una jerarquía entre las operaciones de forma que las que una que está en la parte superior se ejecuta siempre antes que otra que está en otra parte inferior. Los operadores tienen el siguiente orden de precedencia (de mayor a menor):

1. Cambio de signo (-)
2. Multiplicación, división y módulo
3. Suma y resta

Si hay varias operaciones en el mismo nivel, se realizan de izquierda a derecha

Estos niveles de precedencia se pueden "adaptar" a las necesidades del programador mediante el uso de paréntesis. Si hay partes de una expresión que estén entre paréntesis, estas se calcula antes que las que no estén entre paréntesis. Además, los paréntesis se pueden anidar, esto es, colocar dentro de otros paréntesis. En estos casos se realizan primero los cálculos situados en los paréntesis más anidados o internos y se va siguiendo el orden hacia fuera hasta que se terminen todos los cálculos entre paréntesis.

Ejemplos:

Supongamos que tenemos las variables `int a` y `b` con valores `a = 5`, `b = 3` y la variable double `c` con `c = 3.2`

Expresión	Resultado	Desarrollo

`2 + a + 5`	12	`2 + a + 5 =`
		`2 + 5 + 5 =`
		`7 + 5 =`
		`12`

`a + b + c`	11.2	`a + b + c =`
		`5 + 3 + 3.2 =`
		`8 + 3.2 =`
		`11.2`

`2 + a * b`	17	`2 + a * b =`
		`2 + 5 * 3 =`
		`2 + 15 =`
		`17`

`5 * -a / c - 3 * b`	-16.8125	`5 * -5 / 3.2 - 3 * 3 =`
		`-25 / 3.2 - 3 * 3 =`
		`-7.8125 - 3 * 3 =`
		`-7.8125 - 9 =`
		`-16.8125`

`a / b / c`	0.3125	`a / b / c =`
		`5 / 3 / 3.2 =`
		`1 / 3.2 =`
		`0.3125`

`a / (b / c)`	5.333333	`a / (b / c) =`
		`5 / (3 / 3.2) =`
		`5 / 0.9375 =`
		`5.333333`

`2 / (3 * (a - (b + 5)))`	0	`2 / (3 * (a - (b + 5))) =`
		`2 / (3 * (5 - (3 + 5))) =`
		`2 / (3 * (5 - 8)) =`
		`2 / (3 * -3) =`
		`2 / -9 =`
		`0`

`2 / (3 * (a - (b + 5.0)))`	-0.222222	`2 / (3 * (a - (b + 5.0))) =`
		`2 / (3 * (5 - (3 + 5.0))) =`
		`2 / (3 * (5 - 8.0)) =`
		`2 / (3 * -3.0) =`
		`2 / -9.0 =`
		`-0.222222`

`a / b % 5`	1	`a / b % 5 =`
		`5 / 3 % 5 =`
		`1 % 5 =`
		`1`

(Los resultados con decimales son de tipo real, los que no llevan decimales de tipo entero)

12.1.1.- Operadores abreviados de asignación y aritméticos

Una operación muy frecuente que se realiza en programación es tomar el valor de una variable, hacer un cálculo, utilizando ese valor actual, y almacenar el resultado como nuevo valor de la variable. A esta operación se le suele llamar *actualizar la variable*, significando que se calcula de nuevo el valor de una variable para adaptarlo a nuevas circunstancias.

Es ésta una operación tan común que se proporcionan operadores abreviados para realizarlas. Estos operadores son:

- Suma y asignación (`+=`). En primer lugar se calcula el valor de la parte derecha. Una vez calculado, toma ese valor, se lo suma al valor actual de la variable situada en la parte derecha y almacena este resultado en la misma variable. Por ejemplo: `a += 3` significa toma el valor de `a`, súmale `3` y almacena el resultado de nuevo en `a`.
- Resta y asignación (`-=`). Igual que el anterior pero restando. Por ejemplo: `a -= 3` significa toma el valor de `a`, le resta `3` y almacena el resultado de nuevo en `a`.
- Producto y asignación (`*=`). Igual que los anteriores pero multiplicando. Por ejemplo: `a *= 3` significa toma el valor de `a`, lo multiplica por `3` y almacena el resultado de nuevo en `a`.
- División y asignación (`/=`). Igual que los anteriores pero dividiendo. Por ejemplo: `a /= 3` significa toma el valor de `a`, lo divide entre `3` y almacena el resultado de nuevo en `a`.
- Módulo y asignación (`%=`). Igual que los anteriores pero calculando el resto de la división entera. Por ejemplo: `a %= 3` significa toma el valor de `a`, divídelo entre `3` y obtén el resto y almacena éste de nuevo en `a`.

12.2.- Operadores de comparación

Los operadores de comparación toman como operandos dos valores numéricos o de carácter y devuelven valores booleanos (`true` o `false`). Como el nombre indica, los operadores de comparación definen una relación de comparación entre los dos valores. Si los valores a comprobar cumplen la relación, el operador devuelve el valor `true`. Si no , devuelve el valor `false`. Los operadores de comparación son los siguientes:

- Igualdad (`==`). Vale `true` cuando los dos valores son iguales. Ejemplo: `2 == 2` vale `true`, `2 == 1` vale `false`, `-2 == 2` vale `false`, `'a' == 'a'` devuelve `true` y `'a' == 'A'` devuelve `false`.
- Desigualdad (`!=`). Vale `true` cuando los dos valores **NO** son iguales (son distintos). Ejemplo: `2 != 2` vale `false`, `2 != 1` vale `true`, `-2 != 2` vale `true`, `'a' != 'a'` devuelve `false` y `'a' != 'A'` devuelve `true`.
- Mayor que (`>`). Vale `true` cuando el valor a la izquierda es mayor que el valor a la derecha. En el caso de los caracteres, la comparación se realiza comparando los valores numéricos de los códigos Unicode de los caracteres. Ejemplo: `2 > 2` devuelve `false`, `2 > 1` devuelve `true` y `-2 > 2` devuelve `false`, `'a' > 'a'` devuelve `false`, `'a' > 'A'` devuelve `true` (el código numérico del carácter `'a'` (97) es mayor que el de `'A'` (65)).

- Mayor o igual que (>=). Vale `true` cuando el valor de la izquierda es mayor o es igual al de la derecha. Ejemplo: `2 >= 2` devuelve `true`, `2 >= 1` devuelve `true` y `-2 >= 2` devuelve `false`, `'a' >= 'a'` devuelve `true` y `'a' >= 'A'` devuelve `true`.
- Menor que (<). Vale true cuando el valor de la izquierda es menor que el de la derecha. Ejemplo: `2 < 2` devuelve `false`, `2 < 1` devuelve `false` y `-2 < 2` devuelve `true`, `'a' < 'a'` devuelve `false` y `'a' < 'A'` devuelve `false`.
- Mayor que (>). Vale `true` cuando el valor de la izquierda es menor o es igual al de la derecha. Ejemplo: `2 <= 2` devuelve `true`, `2 <= 1` devuelve `false` y `-2 <= 2` devuelve `true`, `'a' <= 'a'` devuelve `true` y `'a' <= 'A'` devuelve `false`.

En el caso de los operadores de comparación, no existe asociación ni precedencia entre ellos ya que los resultados son incompatibles entre si. Por ejemplo, la expresión `2 < 3 < 4` que a primera vista parece correcta, no lo es y produce un error ya que se interpreta como `(2 < 3) < 4` y esto equivale a `true < 4`, lo cual no es comparable y produce un error. Lo mismo ocurriría si la asociación es por la derecha y se evaluara como `2 < (3 < 4)`. Para "unir" varias comparaciones se emplean los operadores lógicos que veremos más adelante.

Si se mezclan operadores de tipo aritmético y lógico, los primeros se realizan antes (tiene precedencia) y posteriormente los segundos. Ejemplo: Si tenemos la expresión `2 + 9 > 5 * 2`, primero se realizarían las operaciones aritméticas, quedando como `11 > 10` y posteriormente se realizarían las operaciones de comparación, quedando el resultado como `true`

Ejemplos:

Supongamos que tenemos las variables `int a` y `b` con valores `a = 5`, `b = 3` y la variable double `c` con `c = 3.2`

Expresión	Resultado	Desarrollo

`a > b`	`true`	`a > b`
		`5 > 3`
		`true`

`a < b`	`false`	`a < b`
		`5 < 3`
		`false`

`a < 8`	`true`	`a < 8`
		`5 < 8`
		`true`

`a >= c`	`true`	`a >= c`
		`5 >= 3.2`
		`true`

`a <= c`	`false`	`5 <= c`
		`5 <= 3.2`
		`false`

`3.2 <= c`	`true`	`3.2 <= c`
`a == b`	`false`	`a == b`

		`5 == 3` `false`

`a != b`	`true`	`a != b`
		`5 != 3`
		`false`

`a == b + 2`	`true`	`a == b + 2`
		`5 == 3 + 2`
		`5 == 5`
		`true`

`a != c + 2`	`true`	`a != c + 2`
		`5 != 3.2 + 2`
		`5 != 5.2`
		`true`

`a + 2 < c * 5`	`true`	`a + 2 < c * 5`
		`5 + 2 < 3.2 * 5`
		`7 < 16`
		`true`

12.3.- Operadores lógicos

Los operadores lógicos combinan valores booleanos (true / false) y devuelven otro valor booleano como resultado. Se utilizan para hacer cálculos de condiciones que implican varios valores booleanos. Los valores sobre los que operan los operadores lógicos son siempre booleanos y devuelven siempre un valor también booleano. Los operadores lógicos son los siguientes:

- No (`!`). Toma un solo operando y devuelve `true` si el operando es `false` y `false` si el operando es `true`. Se puede decir que *niega* (de ahi su nombre) el operando. Ejemplo: `!true` vale `false`, `!false` vale `true`, `!(2 > 3)` vale `true` (`2 > 3` vale `false` porque 2 no es mayor que 3) y `!(2 < 3)` vale `false` (`2 < 3` vale `true` porque 2 es menor que 3).
- Y (&&). Toma dos valores booleanos y devuelve `true` si ambos valen `true` o `false` en caso de que alguno o los dos valga `false`. Ejemplo: `true && true` vale `true`, `true && false` vale `false`, `(2 > 3) && (3 < 4)` vale `false` y `(2 < 3) && (3 < 4)` vale `true`.
- O (`||`). Toma dos valores booleanos y devuelve `true` si alguno de los dos o ambos valen `true` o `false` si ambos valen `false`. Ejemplo: `true || true` vale `true`, `true || false` vale `true`, `(2 > 3) || (3 < 4)` vale `true` y `(2 < 3) || (3 < 4)` vale `true`.

Estos operadores también tienen precedencia entre ellos. El operador `!` tiene máxima prioridad, después el operador && y por último el operador `||`. Si hay varios del mismo tipo se realizan de izquierda a derecha y de forma "perezosa" (lazy evaluation). La forma "perezosa" implica que en cuanto se determina de forma unequívoca el resultado, el resto de la expresión no es evalúa. Por ejemplo si tenemos la expresión (3 > 2) || (4 > 1), primero evaluamos la parte izquierda (3 > 2). Como esto vale true, la parte derecha (4 > 1) no se llega a evaluar siquiera porque no importa lo que valga ya que el resultado será true de todas formas. Esto puede tener importancia cuando el

resultado de la evaluación de la izquierda puede influir de alguna forma en la de la derecha, como veremos en temas siguientes. Por ahora nos es indiferente pero volveremos sobre este asunto más adelante. Asimismo los operadores de comparación y aritméticos tienen precendencia sobre los operadores lógicos.

Ejemplos:

Supongamos que tenemos las variables `int a` y `b` con valores `a = 5`, `b = 3` y la variable double `c` con `c = 3.2`

Expresión	Resultado	Desarrollo
`!true`	`false`	`!true` `false`
`!false`	`true`	`!false` `true`
`true && true`	`true`	`true && true` `true`
`true && false`	`false`	`true && false` `false`
`false && true`	`false`	`false && true` `false`
`false && false`	`false`	`false && false` `false`
`true || true`	`true`	`true || true` `true`
`true || false`	`true`	`true || false` `true`
`false || true`	`true`	`false || true` `true`
`false || false`	`false`	`false || false` `false`

13.- Conversión de Tipos

En Java es posible el convertir un dato de un tipo a un dato de otro tipo, siempre y cuando la conversión tenga sentido y sea posible.

En el caso de los tipos numéricos, hay dos tipos de conversión: ampliación y reducción. En el caso de la ampliación se convierte un dato de un tipo determinado en un dato de un tipo con más rango o capacidad. En el caso de la reducción se hace exactamente lo opuesto. El orden de los tipos es el siguiente, de menor a mayor capacidad:

- byte
- short
- int
- long

- float
- double

La ampliación se hace de forma implícita, esto es, si se asigna un dato de un tipo más reducido a otro de un tipo más amplio, Java realiza la conversión automáticamente y sin avisar. Por ejemplo, el siguiente código:

```
1 int dato1 = 1;
2 long dato2 = 2;
3 dato2 = dato1;
```

Compila sin problemas. En la línea 3 se está haciendo una ampliación del valor entero 1 al valor long 1. Parece lo mismo pero el primero ocupa 32 bits y el segundo 64.

La reducción no se hace de forma implícita y hay que indicarlo de forma explícita. Esto es así porque en la ampliación nunca se pierde información pero en la reducción si puede ocurrir. Por ejemplo, si intentamos convertir el valor 4_000_000_000 (long) a entero, ya que pasamos de un valor de 64 bits a uno de 32. El valor 4_000_000_000 no es un valor válido en un entero y debe ser "recortado", con lo que se pierde información. El resultado es el número -294967296 que se obtiene quedándose con los primeros 32 bits del número de 64, descartando los 32 segundos.

Dado que se produce una pérdida de información y para prevenir problemas accidentales, Java requiere del programador que indique de forma explícita una conversión de reducción. Para ello se usa una expresión llamada *cast* (moldeo). Un cast fuerza una conversión de tipo que de otra forma no se realizaría. El operador de cast tiene la siguiente forma:

`(tipo)valor`

donde `tipo` es el tipo al que se desea convertir el valor `valor`. Si la conversión no es posible Java puede lanzar un error (excepción) durante la ejecución del programa, aunque éste compile. También es posible que produzca un error de compilación si se intenta hacer una conversión sin sentido.

Otra conversión que se puede realizar con cast es la de convertir un carácter a su código unicode. Esto puede hacerse mediante la expresión

`(int)caracter`

donde `caracter` es un literal o variable de tipo `char`.

Ejemplos:

Expresión	Descripción
`int a = (int)1L;`	Conversión de reducción usando cast que convierte el valor long 1 (expresado por el literal 1L) a tipo entero y lo almacena en la variable `a`, que pasará a valer 1 (pero expresado en 32 bits en lugar de 64).
`long l = 1;`	Conversión de amplicación que convierte el valor int 1 a long y lo almacena en la variable `l`
`int b = (int)3.2;`	Conversión de reducción usando cast que convierte el valor real 3.2 a tipo entero. Como el tipo entero no soporta decimales, el valor se recorta a 3 que es el valor que se almacena en `b`.
`char c = 'a';` `int code = (int)c;`	Conversión usando cast que convierte el carácter `'c'` contenido en la variable `c` a entero y almacena el valor (código unicode de `'a'` que

	vale 97) en la variable `code`.

14.- Resumen

En este bloque hemos aprendido los rudimentos del lenguaje Java y de la programación. Ya somos capaces de realizar y ejecutar pequeños programas que realizan tareas simples de cálculo.

15.- Referencias

- JShell Online. https://tryjshell.org.