# Tema 1: Lenguaje JAVA.

## 1.- Introducción.

Cada vez que usamos un ordenador, estamos ejecutando varias aplicaciones que nos permiten realizar ciertas tareas. Por ejemplo, en nuestro día a día, usamos el correo electrónico para enviar y recibir correos, o el navegador para consultar páginas en Internet; ambas actividades son ejemplos de programas que se ejecutan en un ordenador.

Los programas de ordenador deben resolver un problema, para lo cual debemos utilizar de forma inteligente y lógica todos los elementos que nos ofrece el lenguaje. Por eso es importante elegir un lenguaje de programación con el que nos sintamos cómodos porque lo dominemos suficientemente y, por supuesto, porque sepamos que no va a ofrecer limitaciones a la hora de desarrollar aplicaciones para diferentes plataformas.

El lenguaje que vamos a utilizar en este módulo es Java. Es un lenguaje multiplataforma, robusto y fiable. Un lenguaje que reduce la complejidad y se considera dentro de los lenguajes modernos orientados a objetos. Esta unidad nos vamos a adentrar en su sintaxis, vamos a conocer los tipos de datos con los que trabaja, las operaciones que tienen definidas cada uno de ellos, utilizando ejemplos sencillos que nos muestren la utilidad de todo lo aprendido.

Para ello, vamos a tratar sobre cómo se almacenan y recuperan los datos de variables y cadenas en Java, y cómo se gestionan estos datos desde el punto de vista de la utilización de operadores. Trabajar con datos es fundamental en cualquier programa. Aunque ya hayas programado en este lenguaje, échale un vistazo al contenido de esta unidad, porque podrás repasar muchos conceptos.

## 2.- Las variables e identificadores.

Un programa maneja datos para hacer cálculos, presentarlos en informes por pantalla o impresora, solicitarlos al usuario, guardarlos en disco, etc. Para poder manejar esos datos, el programa los guarda en variables.
Una variable es una zona en la memoria del computador con un valor que puede ser almacenado para ser usado más tarde en el programa. Las variables vienen determinadas por:

- Un nombre, que permite al programa acceder al valor que contiene en memoria. Debe ser un identificador válido.
- Un tipo de dato, que especifica qué clase de información guarda la variable en esa zona de memoria
- Un rango de valores que puede admitir dicha variable.

Al nombre que le damos a la variable se le llama **identificador**. Los identificadores permiten nombrar los elementos que se están manejando en un programa. Vamos a ver con más detalle ciertos aspectos sobre los identificadores que debemos tener en cuenta.

## 2.1.- Identificadores.

Un **identificador** en Java es una secuencia ilimitada sin espacios de letras y dígitos Unicode, de forma que **el primer símbolo de la secuencia debe ser una letra, un símbolo de subrayado (_) o el símbolo dólar ($)**. Por ejemplo, son válidos los siguientes identificadores:

```
x5          NUM_MAX     numCuenta
```

En la definición anterior decimos que un identificador es una secuencia ilimitada de caracteres Unicode. Pero… ¿qué es Unicode? Unicode es un código de caracteres o sistema de codificación, un alfabeto que recoge los caracteres de prácticamente todos los idiomas importantes del mundo. Las líneas de código en los programas se escriben usando ese conjunto de caracteres Unicode.

Esto quiere decir que en Java se pueden utilizar varios alfabetos como el Griego, Árabe o Japonés. De esta forma, los programas están más adaptados a los lenguajes e idiomas locales, por lo que son más significativos y fáciles de entender tanto para los programadores que escriben el código, como para los que posteriormente lo tienen que interpretar, para introducir alguna nueva funcionalidad o modificación en la aplicación.

El estándar Unicode originalmente utilizaba 16 bits, pudiendo representar hasta 65.536 caracteres distintos, que es el resultado de elevar dos a la potencia dieciséis. Actualmente Unicode puede utilizar más o menos bits, dependiendo del formato que se utilice: **UTF-8**, **UTF-16** ó **UTF-32. A cada carácter le corresponde unívocamente un número entero perteneciente al intervalo de 0 a 2 elevado a n, siendo n el número de bits utilizados para representar los caracteres. Por ejemplo, la letra** ñ es el entero 164. Además, el código Unicode es "compatible" con el código ASCII, ya que para los caracteres del código ASCII, Unicode asigna como código los mismos 8 bits, a los que les añade a la izquierda otros 8 bits todos a cero. La conversión de un carácter ASCII a Unicode es inmediata.

## 2.2.- Convenios y reglas para nombrar variables.

A la hora de nombrar un identificador existen una serie de normas de estilo de uso generalizado que, no siendo obligatorias, se usan en la mayor parte del código Java. Estas reglas para la nomenclatura de variables son las siguientes:

- **Java distingue las mayúsculas de las minúsculas**. Por ejemplo, `Alumno` y `alumno` son variables diferentes.
- **No se suelen utilizar identificadores que comiencen con «$» o «_»,** además el símbolo del dólar, por convenio, no se utiliza nunca.
- **No se puede utilizar el valor booleano** (`true` o `false`) **ni el valor nulo** (`null`).
- Los **identificadores deben ser lo más descriptivos posibles**. Es mejor usar palabras completas en vez de abreviaturas crípticas. Así nuestro código será más fácil de leer y comprender. En muchos casos también hará que nuestro código se autodocumente. Por ejemplo, si tenemos que darle el nombre a una variable que almacena los datos de un cliente sería recomendable que la misma se llamara algo así como `FicheroClientes` o `ManejadorCliente`, y no algo poco descriptivo como `Cl33`.

Además de estas restricciones, en la siguiente tabla puedes ver otras convenciones, que no siendo obligatorias, sí son recomendables a la hora de crear identificadores en Java.

**Convenciones sobre identificadores en Java**

| Identificador | Convención | Ejemplo |
|---|---|---|
| Nombre de variable. | Comienza por letra minúscula, y si tienen más de una palabra se colocan juntas y el resto comenzando por mayúsculas. | `numAlumnos, suma` |
| Nombre de constante. | En letras mayúsculas, separando las palabras con el guión bajo, por convenio el guión bajo no se utiliza en ningún otro sitio. | `TAM_MAX, PI` |
| Nombre de una clase. | Comienza por letra mayúscula. | `String, MiTipo` |
| Nombre de función. | Comienza con letra minúscula. El resto de palabras con la primera en mayúscula. Evitar el _ | `modificaValor, obtieneValor` |
| | | |

## 2.3.- Palabras reservadas.

Las palabras reservadas, a veces también llamadas palabras clave o keywords , son secuencias de caracteres formadas con letras ASCII cuyo uso se reserva al lenguaje y, por tanto, **no pueden utilizarse para crear identificadores.**

Las palabras reservadas en Java son:

**Palabras clave en Java**

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

Hay palabras reservadas que no se utilizan en la actualidad, como es el caso de **const** y **goto**, que apenas se utilizan en la actual implementación del lenguaje Java. Por otro lado, puede haber otro tipo de palabras o texto en el lenguaje que aunque no sean palabras reservadas tampoco se pueden utilizar para crear

identificadores. Es el caso de **true** y **false** que, aunque puedan parecer palabras reservadas, porque no se pueden utilizar para ningún otro uso en un programa, técnicamente son **literales booleanos**. Igualmente, null es considerado un literal, no una palabra reservada.

Cuando tras haber consultado la documentación de Java aún no tengas seguridad de cómo funciona alguna de sus características, pruébala en tu ordenador, y analiza cada mensaje de error que te dé el compilador para corregirlo. Busca en foros de Internet errores similares para ayudarte de la experiencia de otros usuarios y usuarias.

## 2.4.- Tipos de variables. Constantes I.

En un programa nos podemos encontrar distintos tipos de variables. Las diferencias entre una variable y otra dependerán de varios factores, por ejemplo, el tipo de datos que representan, si su valor cambia o no a lo largo de todo el programa, o cuál es el papel que llevan a cabo en el programa. De esta forma, el lenguaje de programación Java define los siguientes tipos de variables:

a. **Variables de tipos primitivos y variables referencia,** según el tipo de información que contengan. En función de a qué grupo pertenezca la variable, tipos primitivos o tipos referenciados, podrá tomar unos valores u otros, y se podrán definir sobre ella unas operaciones u otras.

b. **Variables y constantes**, dependiendo de si su valor cambia o no durante la ejecución del programa. La definición de cada tipo sería:

   - **Variables**. Sirven para almacenar los datos durante la ejecución del programa, pueden estar formadas por cualquier tipo de dato primitivo o referencia. Su valor puede cambiar varias veces a lo largo de todo el programa.

     **Constantes o variables finales**. Son aquellas variables cuyo valor no cambia a lo largo de todo el programa.

c. **Variables miembro y variables locales**, en función del lugar donde aparezcan en el programa. La definición concreta sería:

   - **Variables miembro**. Son las variables que se crean dentro de una clase, fuera de cualquier método. Pueden ser de tipos primitivos o referencias, variables o constantes. En un lenguaje puramente orientado a objetos como es Java, todo se basa en la utilización de objetos, los cuales se crean usando clases. En la siguiente unidad veremos los distintos tipos de variables miembro que se pueden usar.
   - **Variables locales**. Son las variables que se crean y usan dentro de un método o, en general, dentro de cualquier bloque de código. La variable deja de existir cuando la ejecución del bloque de código o el método finaliza. Al igual que las variables miembro, las variables locales también pueden ser de tipos primitivos o referencias.

# 3.- Los tipos de datos.

En los lenguajes fuertemente tipados, a todo dato (constante, variable o expresión) le corresponde un tipo que es conocido antes de que se ejecute el programa.
El tipo limita el valor de la variable o expresión, las operaciones que se pueden hacer sobre esos valores, y el significado de esas operaciones. Esto es así porque **un tipo de dato no es más que una especificación de los valores que son válidos para esa variable , y de las operaciones que se pueden realizar con ellos.**

Debido a que el tipo de dato de una variable se conoce durante la revisión que hace el compilador para detectar errores, o sea en tiempo de compilación, esta labor es mucho más fácil, ya que no hay que

esperar a que se ejecute el programa para saber qué valores va a contener esa variable. Esto se consigue con un control muy exhaustivo de los tipos de datos que se utilizan, lo cual tiene sus ventajas e inconvenientes, por ejemplo cuando se intenta asignar un valor de un tipo, a una variable de otro tipo. Sin embargo, en Java, puede haber conversión entre ciertos tipos de datos, como veremos posteriormente.
Ahora no es el momento de entrar en detalle sobre la conversión de tipos, pero sí debemos conocer con exactitud de qué tipos de datos dispone el lenguaje Java. Ya hemos visto que las variables, según la información que contienen, se pueden dividir en variables de tipos primitivos y variables referencia. Pero ¿qué es un tipo de dato primitivo? ¿Y un tipo referencia? Esto es lo que vamos a ver a continuación. Los tipos de datos en Java se dividen principalmente en dos categorías:

- **Tipos de datos sencillos o primitivos**. Representan valores simples que vienen predefinidos en el lenguaje; contienen valores únicos, como por ejemplo un carácter o un número.
- **Tipos de datos referencia.** Se definen con un nombre o referencia (puntero) que contiene la dirección en memoria de un valor o grupo de valores. Dentro de este tipo tenemos por ejemplo los vectores o arrays, que son una serie de elementos del mismo tipo, o las clases, que son los modelos o plantillas a partir de los cuales se crean los objetos.

# 3.1.- Tipos de datos primitivos I.

Los tipos primitivos son aquéllos datos sencillos que constituyen los tipos de información más habituales: números, caracteres y valores lógicos o booleanos. Al contrario que en otros lenguajes de programación orientados a objetos, **en Java no son objetos**.

Una de las mayores ventajas de tratar con tipos primitivos en lugar de con objetos, es que el compilador de Java puede optimizar mejor su uso. Otra importante característica, es que cada uno de los tipos primitivos tiene **idéntico** tamaño y comportamiento en todas las versiones de Java y para cualquier tipo de ordenador. Esto quiere decir que no debemos preocuparnos de cómo se representarán los datos en distintas plataformas, y asegura la **portabilidad** de los programas, a diferencia de lo que ocurre con otros lenguajes. Por ejemplo, el tipo **int** siempre se representará con 32 bits, con signo, y en el formato de representación complemento a 2, en cualquier plataforma que soporte Java.

Hay una peculiaridad en los tipos de datos primitivos, y es que el tipo de dato char es considerado por el compilador como un tipo numérico, ya que los valores que guarda son el código Unicode correspondiente al carácter que representa, no el carácter en sí, por lo que puede operarse con caracteres como si se tratara de números enteros.

Por otra parte, a la hora de elegir el tipo de dato que vamos a utilizar ¿qué criterio seguiremos para elegir un tipo de dato u otro? Pues deberemos tener en cuenta cómo es la información que hay que guardar, si es de tipo texto, numérico, … y, además, qué rango de valores puede alcanzar. En este sentido, hay veces que aunque queramos representar un número sin decimales, tendremos que utilizar datos de tipo real.

Por ejemplo, el tipo de dato **int** no podría representar la población mundial del planeta, ya que el valor máximo que alcanza es de 2.147.483.647, siendo éste el número máximo de combinaciones posibles con 32 bits, teniendo en cuenta que la representación de los números enteros en Java utiliza complemento a 2. Si queremos representar el valor correspondiente a la población mundial del planeta, cerca de 7 mil millones de habitantes, tendríamos que utilizar al menos un tipo de dato **long**, o si tenemos problemas de espacio un tipo **float**. Sin embargo, los tipos reales tienen otro problema: la **precisión**. Vamos a ver más sobre ellos en el siguiente apartado.

## 3.1.1.- Tipos de datos primitivos II.

| Tipo de dato | Descripción | Tamaño | Rango de valores |
|---|---|---|---|
| **short** | Número entero corto | 2 bytes | -32768 a 32767 |
| **int** | Número Entero con signo | 4 bytes | $-2^{31}$ a $2^{31}-1$ |
| **long** | Número Entero largo con signo | 8 bytes | $-2^{63}$ a $2^{63}-1$ |
| **float** | Número coma flotante simple precisión | 4 bytes | $3.4x10^{-38}$ a $3.4x10^{+45}$ |
| **double** | Número coma flotante doble precisión | 8 bytes | $1.8x10^{-308}$ a $1.8x10^{+324}$ |
| **char** | Carácter Unicode entre comillas simples | 2 bytes | |
| **boolean** | Verdadero o Falso | 1 byte | true o false |

El tipo de dato real permite representar cualquier número con decimales. Al igual que ocurre con los enteros, la mayoría de los lenguajes definen más de un tipo de dato real, en función del número de bits usado para representarlos. Cuanto mayor sea ese número:

- **Más grande podrá ser el número real representado en valor absoluto**
- **Mayor será la precisión** de la parte decimal

Entre cada dos números reales cualesquiera, siempre tendremos infinitos números reales, por lo que **la mayoría de ellos los representaremos de forma aproximada.** Por ejemplo, en la aritmética convencional, cuando dividimos 10 entre 3, el resultado es 3.3333333…, con la secuencia de 3 repitiéndose infinitamente. En el ordenador sólo podemos almacenar un número finito de bits, por lo que el almacenamiento de un número real será siempre una aproximación.

Los números reales se representan en coma flotante, que consiste en trasladar la coma decimal a la primera cifra significativa del valor, con objeto de poder representar el máximo de números posible.

En Java las variables de tipo **float** se emplean para representar los números en coma flotante de simple precisión de 32 bits, de los cuales 24 son para la mantisa y 8 para el exponente. La mantisa es un valor entre -1.0 y 1.0 y el exponente representa la potencia de 2 necesaria para obtener el valor que se quiere representar. Por su parte, las variables tipo **double** representan los números en coma flotante de doble precisión de 64 bits, de los cuales 53 son para la mantisa y 11 para el exponente.

La mayoría de los programadores en Java emplean el tipo **double** cuando trabajan con datos de tipo real, es una forma de asegurarse de que los errores cometidos por las sucesivas aproximaciones sean menores. De hecho, Java considera los valores en coma flotante como de tipo **double** por defecto.

Con el objetivo de conseguir la máxima portabilidad de los programas, Java utiliza el estándar internacional IEEE 754 para la representación interna de los números en coma flotante, que es una forma de asegurarse de que el resultado de los cálculos sea el mismo para diferentes plataformas.

# 3.2.- Declaración e inicialización.

Llegados a este punto cabe preguntarnos ¿cómo se crean las variables en un programa? ¿Qué debo hacer antes de usar una variable en mi programa? Pues bien, como podrás imaginar, debemos crear las variables antes de poder utilizarlas en nuestros programas, indicando qué nombre va a tener y qué tipo de información va a almacenar, en definitiva, debemos **declarar la variable**.

Las variables se pueden declarar en cualquier bloque de código, dentro de llaves. Y lo hacemos indicando su identificador y el tipo de dato, separadas por comas si vamos a declarar varias a la vez, por ejemplo:

```
int numAlumnos = 15;
double radio = 3.14, importe = 102.95;
```

De esta forma, estamos declarando **numAlumnos** como una variable de tipo **int**, y otras dos variables **radio** e **importe** de tipo double. Aunque no es obligatorio, hemos aprovechado la declaración de las variables para inicializarlas a los valores 15, 3.14 y 102.95 respectivamente.

Si la variable va a permanecer inalterable a lo largo del programa, la declararemos como **constante**, utilizando la palabra reservada **final** de la siguiente forma:

```
final double PI = 3.1415926536;
```

En ocasiones puede que al declarar una variable no le demos valor, ¿qué crees que ocurre en estos casos? Pues que el compilador le asigna un valor por defecto, aunque depende del tipo de variable que se trate:

- Las **variables miembro** sí se inicializan automáticamente, si no les damos un valor. Cuando son de tipo numérico, se inicializan por defecto a `0`, si don de tipo carácter, se inicializan al carácter `null` (\0), si son de tipo `boolean` se les asigna el valor por defecto `false`, y si son tipo referenciado se inicializan a `null`.
- Las **variables locales** no se inicializan automáticamente. Debemos asignarles nosotros un valor antes de ser usadas, ya que si el compilador detecta que la variable se usa antes de que se le asigne un valor, produce un error. Por ejemplo en este caso:

```
int p;
int q = p; // error
```

Y también en este otro, ya que se intenta usar una variable local que podría no haberse inicializado:

```
int p;
if (. . . )
     p = 5 ;
int q = p; // error
```

En el ejemplo anterior la instrucción **if** hace que si se cumple la condición que hay entre paréntesis (cualquiera que indiquemos), entonces el programa asignará el valor 5 a la variable **p**; sino se cumple la condición, **p** quedará sin inicializar. Pero si **p** no se ha inicializado, no tendría valor para asignárselo a **q**. Por ello, el compilador detecta ese posible problema y produce un error del tipo **"La variable podría no haber sido inicializada"**, independientemente de si se cumple o no la condición del **if**.

Además de los ocho tipos de datos primitivos que ya hemos descrito, Java proporciona un tratamiento especial a los textos o cadenas de caracteres mediante el tipo de dato **String**. Java crea automáticamente un nuevo objeto de tipo **String** cuando se encuentra una cadena de caracteres encerrada entre comillas dobles. En realidad se trata de objetos, y por tanto son tipos referenciados, pero se pueden utilizar de forma sencilla como si fueran variables de tipos primitivos:

```
String mensaje;
mensaje= "El primer programa";
```

Hemos visto qué son las variables, cómo se declaran y los tipos de datos que pueden adoptar. Anteriormente hemos visto un ejemplo de creación de variables, en esta ocasión vamos a crear más variables, pero de distintos tipos primitivos y los vamos a mostrar por pantalla.

Para mostrar por pantalla un mensaje utilizamos **System.out**, conocido como la salida estándar del programa. Este método lo que hace es escribir un conjunto de caracteres a través de la línea de comandos. En Netbeans esta línea de comandos aparece en la parte inferior de la pantalla. Podemos utilizar **System.out.print** o **System.out.println**. En el segundo caso lo que hace el método es que justo después de escribir el mensaje, sitúa el cursor al principio de la línea siguiente. El texto en color gris que aparece entre caracteres // son comentarios que permiten documentar el código, pero no son tenidos en cuenta por el compilador y, por tanto, no afectan a la ejecución del programa.

# 4.- Literales de los tipos primitivos.

Un **literal**, **valor literal** o **constante literal** es un valor concreto para los tipos de datos primitivos del lenguaje, el tipo **String** o el tipo **null**.

Los **literales booleanos** tienen dos únicos valores que puede aceptar el tipo: **true** y **false**. Por ejemplo, con la instrucción **boolean encontrado = true;** estamos declarando una variable de tipo booleana a la cual le asignamos el valor literal **true**.

Los **literales enteros** se pueden representar en tres notaciones:

- **Decimal**: por ejemplo `20`. Es la forma más común.
- **Octal**: por ejemplo `024`. Un número en octal siempre empieza por cero, seguido de dígitos octales (del 0 al 7).
- **Hexadecimal**: por ejemplo `0x14`. Un número en hexadecimal siempre empieza por `0x` seguido de dígitos hexadecimales (del 0 al 9, de la 'a' a la 'f' o de la 'A' a la 'F').

Las constantes literales de tipo **long** se le debe añadir detrás una **l** ó **L**, por ejemplo **873L**, si no se considera por defecto de tipo **int**. Se suele utilizar **L** para evitar la confusión de la ele minúscula con 1.

Los **literales reales** o en coma flotante se expresan con coma decimal o en notación científica, o sea, seguidos de un exponente **e** ó **E**. El valor puede finalizarse con una f o una F para indica el formato float o con una d o una D para indicar el formato **double** (por defecto es **double**). Por ejemplo, podemos representar un mismo literal real de las siguientes formas: **13.2, 13.2D, 1.32e1, 0.132E2**. Otras constantes literales reales son por ejemplo: **.54, 31.21E-5, 2.f, 6.022137e+23f, 3.141e-9d.**

Un **literal carácter** puede escribirse como un carácter entre comillas simples como **'a'**, **'ñ'**, **'Z'**, **'p'**, etc. o por su código de la tabla Unicode, anteponiendo la secuencia de escape '\' si el valor lo ponemos en octal o '**\u**' si ponemos el valor en hexadecimal. Por ejemplo, si sabemos que tanto en ASCII como en Unicode, la letra A (mayúscula) es el símbolo número 65, y que 65 en octal es 101 y 41 en hexadecimal, podemos representar esta letra como **'\101'** en octal y **'\u0041'** en hexadecimal. Existen unos caracteres especiales que se representan utilizando secuencias de escape:

**Secuencias de escape en Java**

| Secuencia de escape | Significado | Secuencia de escape | Significado |
|---|---|---|---|
| \b | Retroceso | \r | Retorno de carro |
| \t | Tabulador | \'' | Carácter comillas dobles |
| \n | Salto de línea | \' | Carácter comillas simples |
| \f | Salto de página | \\ | Barra diagonal |

**Normalmente, los objetos en Java deben ser creados con la orden new. Sin embargo, los literales String no lo necesitan ya que son objetos que se crean implícitamente por Java.**

Los **literales de cadenas de caracteres** se indican entre comillas dobles. En el ejemplo anterior "**El primer programa**" es un literal de tipo cadena de caracteres. Al construir una cadena de caracteres se puede incluir cualquier carácter Unicode excepto un carácter de retorno de carro, por ejemplo en la siguiente instrucción utilizamos la secuencia de escape \'' para escribir dobles comillas dentro del mensaje:

```
String texto = "Juan dijo: \"Hoy hace un día fantástico…\"";
```

En el ejemplo anterior de tipos enumerados ya estábamos utilizando secuencias de escape, para introducir un salto de línea en una cadena de caracteres, utilizando el carácter especial**\n**.

# 5.- Operadores y expresiones.

Los **operadores** llevan a cabo operaciones sobre un conjunto de datos u operandos, representados por literales y/o identificadores. Los operadores pueden ser unarios, binarios o terciarios, según el número de operandos que utilicen sean uno, dos o tres, respectivamente. Los operadores actúan sobre los tipos de datos primitivos y devuelven también un tipo de dato primitivo.

Los operadores se combinan con los literales y/o identificadores para formar expresiones. Una **expresión** es una combinación de operadores y operandos que se evalúa produciendo un único resultado de un tipo determinado.

El resultado de una expresión puede ser usado como parte de otra expresión o en una sentencia o instrucción. Las expresiones, combinadas con algunas palabras reservadas o por sí mismas, forman las llamadas **sentencias** o **instrucciones**.
Por ejemplo, pensemos en la siguiente expresión Java:

```
i + 1
```

Con esta expresión estamos utilizando un operador aritmético para sumarle una cantidad a la variable i, pero es necesario indicar al programa qué hacer con el resultado de dicha expresión:

```
suma = i + 1;
```

Que lo almacene en una variable llamada **suma**, por ejemplo. En este caso ya tendríamos una acción completa, es decir, una sentencia o instrucción.

Más ejemplos de sentencias o instrucciones los tenemos en las declaraciones de variables, vistas en apartados anteriores, o en las estructuras básicas del lenguaje como sentencias condicionales o bucles, que veremos en unidades posteriores.

Como curiosidad comentar que las expresiones de asignación, al poder ser usadas como parte de otras asignaciones u operaciones, son consideradas tanto expresiones en sí mismas como sentencias.

## 5.1.- Operadores aritméticos.

Los operadores aritméticos son aquellos operados que combinados con los operandos forman expresiones matemáticas o aritméticas.

**Operadores aritméticos básicos**

| Operador | Operación Java | Expresión Java | Resultado |
|---|---|---|---|
| - | Operador unario de cambio de signo | -10 | -10 |
| + | Adición | 1.2 + 9.3 | 10.5 |
| - | Sustracción | 312.5 – 12.3 | 300.2 |
| * | Multiplicación | 1.7 * 1.2 | 1.02 |
| / | División (entera o real) | 0.5 / 0.2 | 2.5 |
| **%** | Resto de la división entera | 25 % 3 | 1 |

El resultado de este tipo de expresiones depende de los operandos que utilicen:

**Resultados de las operaciones aritméticas en Java**

| Tipo de los operandos | Resultado |
|---|---|
| Un operando de tipo `long` y ninguno real (`float` o `double`) | `long` |
| Ningún operando de tipo `long` ni real (`float` o `double`) | `int` |
| Al menos un operando de tipo `double` | `double` |
| Al menos un operando de tipo `float` y ninguno `double` | `float` |

Otro tipo de operadores aritméticos son los operadores unarios incrementales y decrementales. Producen un resultado del mismo tipo que el operando, y podemos utilizarlos con**notación prefija**, si el operador aparece antes que el operando, o **notación postfija**, si el operador aparece después del operando. En la tabla puedes un ejemplo de de utilización de cada operador incremental.

| Operadores incrementales en Java | | |
|---|---|---|
| **Tipo operador** | **Expresión Java** | |
| **++ (incremental)** | Prefija:<br>x=3;<br>y=++x;<br>// x vale 4 e y vale 4 | Postfija:<br>x=3;<br>y=x++;<br>// x vale 4 e y vale 3 |
| **--(decremental)** | 5-- // el resultado es 4 | |

## 5.2.- Operadores de asignación.

El principal operador de esta categoría es el operador asignación "=", que permite al programa darle un valor a una variable, y ya hemos utilizado varias ocasiones en esta unidad. Además de este operador, Java proporciona otros operadores de asignación combinados con los operadores aritméticos, que permiten abreviar o reducir ciertas expresiones.

Por ejemplo, el operador "+=" suma el valor de la expresión a la derecha del operador con el valor de la variable que hay a la izquierda del operador, y almacena el resultado en dicha variable. En la siguiente tabla se muestran todos los operadores de asignación compuestos que podemos utilizar en Java

| Operadores de asignación combinados en Java | | |
|---|---|---|
| **Operador** | **Ejemplo en Java** | **Expresión equivalente** |
| `+=` | `op1 += op2` | `op1 = op1 + op2` |
| `-=` | `op1 -= op2` | `op1 = op1 - op2` |
| `*=` | `op1 *= op2` | `op1 = op1 * op2` |
| `/=` | `op1 /= op2` | `op1 = op1 / op2` |
| `%=` | `op1 %= op2` | `op1 = op1 % op2` |

## 5.4.- Operadores de relación.

Los operadores relacionales se utilizan para comparar datos de tipo primitivo (numérico, carácter y booleano). El resultado se utilizará en otras expresiones o sentencias, que ejecutarán una acción u otra en función de si se cumple o no la relación.

Estas expresiones en Java dan siempre como resultado un valor booleano true o false. En la tabla siguiente aparecen los operadores relacionales en Java.

**Operadores relacionales en Java**

| Operador | Ejemplo en Java | Significado |
|---|---|---|
| == | op1 == op2 | op1 igual a op2 |
| != | op1 != op2 | op1 distinto de op2 |
| > | op1 > op2 | op1 mayor que op2 |
| < | op1 < op2 | op1 menor que op2 |
| >= | op1 >= op2 | op1 mayor o igual que op2 |
| <= | op1 <= op2 | op1 menor o igual que op2 |

## 5.5.- Operadores lógicos.

Los operadores lógicos realizan operaciones sobre valores booleanos, o resultados de expresiones relacionales, dando como resultado un valor booleano.

Los operadores lógicos los podemos ver en la tabla que se muestra a continuación. Existen ciertos casos en los que el segundo operando de una expresión lógica no se evalúa para ahorrar tiempo de ejecución, porque con el primero ya es suficiente para saber cuál va a ser el resultado de la expresión.

Por ejemplo, en la expresión **a && b** si **a** es falso, no se sigue comprobando la expresión, puesto que ya se sabe que la condición de que ambos sean verdadero no se va a cumplir. En estos casos es más conveniente colocar el operando más propenso a ser falso en el lado de la izquierda. Igual ocurre con el operador ||, en cuyo caso es más favorable colocar el operando más propenso a ser verdadero en el lado izquierdo.

**Operadores lógicos en Java**

| Operador | Ejemplo en Java | Significado |
|---|---|---|
| ! | !op | Devuelve true si el operando es false y viceversa. |
| & | op1 & op2 | Devuelve true si op1 y op2 son true |
| \| | op1 \| op2 | Devuelve true si op1 u op2 son true |
| ^ | op1 ^ op2 | Devuelve true si sólo uno de los operandos es true |
| && | op1 && op2 | Igual que &, pero si op1 es false ya no se evalúa op2 |
| \|\| | op1 \|\| op2 | Igual que \|, pero si op1 es true ya no se evalúa op2 |

```java
public class operadoreslogicos {
  public static void main(String[] args) {
    System.out.println("OPERADORES LÓGICOS");

    System.out.println("Negacion:\n ! false es : " + (! false));
    System.out.println(" ! true es : " + (! true));

    System.out.println("Operador AND (&):\n false & false es : " + (false & false));
    System.out.println(" false & true es : " + (false & true));
    System.out.println(" true & false es : " + (true & false));
    System.out.println(" true & true es : " + (true & true));

    System.out.println("Operador OR (|):\n false | false es : " + (false | false));
    System.out.println(" false | true es : " + (false | true));
    System.out.println(" true | false es : " + (true | false));
    System.out.println(" true | true es : " + (true | true));

    System.out.println("Operador OR Exclusivo (^):\n false ^ false es : " + (false ^ false));
    System.out.println(" false ^ true es : " + (false ^ true));
    System.out.println(" true ^ false es : " + (true ^ false));
    System.out.println(" true ^ true es : " + (true ^ true));

    System.out.println("Operador &&:\n false && false es : " + (false && false));
    System.out.println(" false && true es : " + (false && true));
    System.out.println(" true && false es : " + (true && false));
    System.out.println(" true && true es : " + (true && true));

    System.out.println("Operador ||:\n false || false es : " + (false || false));
    System.out.println(" false || true es : " + (false || true));
    System.out.println(" true || false es : " + (true || false));
    System.out.println(" true || true es : " + (true || true));
  } // fin main
} // fin operadoreslogicos
```

# 5.6.- Precedencia de operadores.

El orden de precedencia de los operadores determina la secuencia en que deben realizarse las operaciones cuando en una expresión intervienen operadores de distinto tipo.

Las reglas de precedencia de operadores que utiliza Java coinciden con las reglas de las expresiones del álgebra convencional. Por ejemplo:

- La multiplicación, división y resto de una operación se evalúan primero. Si dentro de la misma expresión tengo varias operaciones de este tipo, empezaré evaluándolas de izquierda a derecha.
- La suma y la resta se aplican después que las anteriores. De la misma forma, si dentro de la misma expresión tengo varias sumas y restas empezaré evaluándolas de izquierda a derecha.

A la hora de evaluar una expresión es necesario tener en cuenta la **asociatividad** de los operadores. La asociatividad indica qué operador se evalúa antes, en condiciones de igualdad de precedencia. Los operadores de asignación, el operador condicional (`?:`), los operadores incrementales (`++`, `--`) y el casting son asociativos por la derecha. El resto de operadores son asociativos por la izquierda, es decir, que se empiezan a calcular en el mismo orden en el que están escritos: de izquierda a derecha. Por ejemplo, en la expresión siguiente:

```
10 / 2 * 5
```

Realmente la operación que se realiza es `(10 / 2 ) * 5`, porque ambos operadores, división y multiplicación, tienen igual precedencia y por tanto se evalúa primero el que antes nos encontramos por la izquierda, que es la división. El resultado de la expresión es `25`. Si fueran asociativos por la derecha, puedes comprobar que el resultado sería diferente, primero multiplicaríamos `2 * 5` y luego dividiríamos entre `10`, por lo que el resultado sería 1. En esta otra expresión:

```
x = y = z = 1
```

Realmente la operación que se realiza es `x = (y = (z = 1))`. Primero asignamos el valor de `1` a la variable `z`, luego a la variable `y`, para terminar asignando el resultado de esta última asignación a `x`. Si el operador

asignación fuera asociativo por la izquierda esta operación no se podría realizar, ya que intentaríamos asignar a `x` el valor de `y`, pero `y` aún no habría sido inicializada.

En la tabla se detalla el orden de precedencia y la asociatividad de los operadores que hemos comentado en este apartado. Los operadores se muestran de mayor a menor precedencia.

**Orden de precedencia de operadores en Java**

| Operador | Tipo | Asociatividad |
|---|---|---|
| `++ --` | Unario, notación postfija | Derecha |
| `++ -- + - (cast) ! ~` | Unario, notación prefija | Derecha |
| `* / %` | Aritméticos | Izquierda |
| `+ -` | Aritméticos | Izquierda |
| `<< >> >>>` | Bits | Izquierda |
| `< <= > >=` | Relacionales | Izquierda |
| `== !=` | Relacionales | Izquierda |
| `&` | Lógico, Bits | Izquierda |
| `^` | Lógico, Bits | Izquierda |
| `\|` | Lógico, Bits | Izquierda |
| `&&` | Lógico | Izquierda |
| `\|\|` | Lógico | Izquierda |
| `?:` | Operador condicional | Derecha |
| `= += -= *= /= %=` | Asignación | Derecha |

# 6.- Conversión de tipo.

Imagina que queremos dividir un número entre otro ¿tendrá decimales el resultado de esa división? Podemos pensar que siempre que el denominador no sea divisible entre el divisor, tendremos un resultado con decimales, pero no es así. Si el denominador y el divisor son variables de tipo entero, el resultado será entero y no tendrá decimales. Para que el resultado tenga decimales necesitaremos hacer una **conversión de tipo**.

Las conversiones de tipo se realizan para hacer que el resultado de una expresión sea del tipo que nosotros deseamos, en el ejemplo anterior para hacer que el resultado de la división sea de tipo real y, por ende, tenga decimales. Existen dos tipos de conversiones:

- **Conversiones automáticas**. Cuando a una variable de un tipo se le asigna un valor de otro tipo numérico con menos bits para su representación, se realiza una conversión automática. En ese caso, el valor se dice que es promocionado al tipo más grande (el de la variable), para poder hacer la asignación. También se realizan conversiones automáticas en las operaciones aritméticas, cuando estamos utilizando valores de distinto tipo, el valor más pequeño se promociona al valor más grande, ya que el tipo mayor siempre podrá representar cualquier valor del tipo menor (por ejemplo, de **int** a **long** o de **float** a **double**).

- **Conversiones explícitas**. Cuando hacemos una conversión de un tipo con más bits a un tipo con menos bits. En estos casos debemos indicar que queremos hacer la conversión de manera expresa, ya que se puede producir una pérdida de datos y hemos de ser conscientes de ello. Este tipo de conversiones se realiza con el **operador cast**. El operador**cast** es un operador unario que se forma colocando delante del valor a convertir el tipo de dato entre paréntesis. Tiene la misma precedencia que el resto de operadores unarios y se asocia de izquierda a derecha.

Debemos tener en cuenta que **un valor numérico nunca puede ser asignado a una variable de un tipo menor en rango, si no es con una conversión explícita**. Por ejemplo:

```
int a;
byte b;
a = 12;                 // no se realiza conversión alguna
b = 12;                 // se permite porque 12 está dentro
                        // del rango permitido de valores para b
b = a;                  // error, no permitido (incluso aunque
                        // 12 podría almacenarse en un byte)
byte b = (byte) a;              // Correcto, forzamos conversión explícita
```

En el ejemplo anterior vemos un caso típico de error de tipos, ya que estamos intentando asignarle a **b** el valor de **a**, siendo **b** de un tipo más pequeño. Lo correcto es promocionar **a** al tipo de datos **byte**, y entonces asignarle su valor a la variable **b**.

# 7.- Comentarios.

Los comentarios son muy importantes a la hora de describir qué hace un determinado programa. A lo largo de la unidad los hemos utilizado para documentar los ejemplos y mejorar la comprensión del código. Para lograr ese objetivo, es normal que cada programa comience con unas líneas de comentario que indiquen, al menos, una breve descripción del programa, el autor del mismo y la última fecha en que se ha modificado.

Todos los lenguajes de programación disponen de alguna forma de introducir comentarios en el código. En el caso de Java, nos podemos encontrar los siguientes tipos de comentarios:

- **Comentarios de una sola línea**. Utilizaremos el delimitador `//` para introducir comentarios de sólo una línea.

```
// comentario de una sola línea
```

- **Comentarios de múltiples líneas**. Para introducir este tipo de comentarios, utilizaremos una barra inclinada y un asterisco (`/*`), al principio del párrafo y un asterisco seguido de una barra inclinada (`*/`) al final del mismo.

```
/* Esto es un comentario
de varias líneas */
```

- **Comentarios Javadoc**. Utilizaremos los delimitadores /** y */. Al igual que con los comentarios tradicionales, el texto entre estos delimitadores será ignorado por el compilador. Este tipo de comentarios se emplean para generar documentación automática del programa. A través del programa javadoc, incluido en JavaSE , se recogen todos estos comentarios y se llevan a un documento en formato .html.

# Anexo I.- Conversión de tipos de datos en Java.

**Tabla de Conversión de Tipos de Datos Primitivos**

| | | Tipo destino | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **boolean** | **char** | **byte** | **short** | **int** | **long** | **float** | **double** |
| **Tipo origen** | boolean | - | N | N | N | N | N | N | N |
| | char | N | - | C | C | CI | CI | CI | CI |
| | byte | N | C | - | CI | CI | CI | CI | CI |
| | short | N | C | C | - | CI | CI | CI | CI |
| | int | N | C | C | C | - | CI | CI* | CI |
| | long | N | C | C | C | C | - | CI* | CI* |
| | float | N | C | C | C | C | C | - | CI |
| | double | N | C | C | C | C | C | C | - |

Explicación de los símbolos utilizados:

**N:** Conversión no permitida (un **boolean** no se puede convertir a ningún otro tipo y viceversa).
**CI:** Conversión implícita o automática. Un asterisco indica que puede haber posible pérdida de datos.
**C:** Casting de tipos o conversión explícita.

El asterisco indica que puede haber una posible pérdida de datos, por ejemplo al convertir un número de tipo **int** que usa los 32 bits posibles de la representación, a un tipo **float**, que también usa 32 bits para la representación, pero 8 de los cuales son para el exponente.

En cualquier caso, las conversiones de números en coma flotante a números enteros siempre necesitarán un **Casting**, y deberemos tener mucho cuidado debido a la pérdida de precisión que ello supone.

**Reglas de Promoción de Tipos de Datos**

Cuando en una expresión hay datos o variables de distinto tipo, el compilador realiza la promoción de unos tipos en otros, para obtener como resultado el tipo final de la expresión. Esta promoción de tipos se hace siguiendo unas reglas básicas en base a las cuales se realiza esta promoción de tipos, y resumidamente son las siguientes:

- Si uno de los operandos es de tipo **`double`**, el otro es convertido a **`double`**.
- En cualquier otro caso:
  - Si el uno de los operandos es **`float`**, el otro se convierte a **`float`**
  - Si uno de los operandos es **`long`**, el otro se convierte a **`long`**
  - Si no se cumple ninguna de las condiciones anteriores, entonces ambos operandos son convertidos al tipo **`int`**.

## Tabla sobre otras consideraciones con los Tipos de Datos

### Conversiones de números en Coma flotante (`float`, `double`) a enteros (`int`)

Cuando convertimos números en coma flotante a números enteros, la parte decimal se trunca (redondeo a cero). Si queremos hacer otro tipo de redondeo, podemos utilizar, entre otras, las siguientes funciones:

- **Math.round(num):** Redondeo al siguiente número entero.
- **Math.ceil(num):** Mínimo entero que sea mayor o igual a num.
- **Math.floor(num):** Entero mayor, que sea inferior o igual a num.

```
double num=3.5;

x=Math.round(num); //x = 4
y=Math.ceil(num);  //y = 4
z=Math.floor(num); //z = 3
```

### Conversiones entre caracteres (`char`) y enteros (`int`)

Como un tipo **char** lo que guarda en realidad es el código **Unicode** de un carácter, los caracteres pueden ser considerados como números enteros sin signo.

**Ejemplo:**

```
int num;
char c;
num =(int)'A';
   //num vale 65

c =(char) 65;
   //c vale 'A'

c=(char)((int)'A'+1);
 //c = 'B'
```

### Conversiones de tipo con cadenas de caracteres (`String`)

Para convertir cadenas de texto a otros tipos de datos se utilizan las siguientes funciones:

```
num=Byte.parseByte(cad);
num=Short.parseShort(cad);
num=Integer.parseInt(cad);
num=Long.parseLong(cad);
num=Float.parseFloat(cad);
num=Double.parseDouble(cad);
```

Por ejemplo, si hemos leído de teclado un número que está almacenado en una variable de tipo **String** llamada **cadena**, y lo queremos convertir al tipo de datos **byte**, haríamos lo siguiente:

```
byte
n=Byte.parseByte(cadena);
```