# Apunte de clase 4 - Sintaxis

En el apunte de clases 2, hicimos ya una breve identificación del Lenguaje de Programación Java en cuanto a sus características generales específicas. Mencionamos allí que Java es un lenguaje de programación basado en *Objetos*, *Case Sensitive*, que utiliza *Llaves* para delimitar los bloques de código y que finaliza las sentencias de programación con un *Punto y coma*. Esto no es poco decir porque en un simple párrafo estamos delineando los elementos fundantes del lenguaje.

En este apunte de clases vamos a sumar los conceptos iniciales y necesarios para poder programar en Java como son los *Tipos de datos*, *Operaciones* tanto de asignación como operaciones propiamente dichas, *Estructuras de control* y finalizaremos el presente apunte con alguna herramienta introductoria para *Lectura o Ingreso de datos*. Allá vamos.

## Tipos, Variables y Asignaciones

Todo programa Java necesitará en algún momento recibir, operar, almacenar y mostrar o devolver valores o datos. Para ello como en todos los lenguajes de programación vistos hasta aquí vamos a utilizar variables. Sin embargo, con java será la primera vez que nos encontremos frente a un lenguaje de programación *tipado*, es decir que toda variable tiene un tipo de datos y este tipo de datos en la mayoría de los casos debe ser especificado por el programador y no puede cambiar en el futuro.

Estas variables ocuparán en memoria un cierto número de bytes, que depende del tipo de valor del que se trate (por ejemplo, en general en distintos lenguajes de programación un valor de tipo entero ocupa entre uno y ocho bytes en memoria, un valor de tipo real o de coma flotante, con punto y parte decimal, ocupa entre cuatro y ocho bytes, y un caracter ocupa uno o dos bytes dependiendo de la codificación empleada) y Java no es diferente en este aspecto.

Por ello antes de poder utilizar una variable deberemos *declarar* esa variable y qué hacemos cuando declaramos una variable, específicamente unimos los tres componentes de esta: el nombre o identificador a partir del cual luego nos referiremos a la variable por un lado, el tipo de datos que la variable puede contener por el otro y finalmente, aunque transparente para nosotros, la memoria que esa variable dispondrá para almacenar los valores que asociemos a ella.

Bien vamos por partes, cuáles son los tipos de variables existentes en java:

- Tipos enteros

| Tipo | Descripción | Bytes ocupados | Literales | Rango de Valores |
|---|---|---|---|---|
| `byte` | número entero de un byte | 1 | número entre -128 y 127 | [-128 a 127] |
| `short` | número entero de dos bytes | 2 | número entero en el rango | [-32768 a 32767] |

| Tipo | Descripción | Bytes ocupados | Literales | Rango de Valores |
|---|---|---|---|---|
| `int` | número entero de cuatro bytes | 4 | número entre en el rango | [-2^31 a 2^31-1] |
| `long` | número entero de ocho bytes | 8 | número entero | |

- Tipos decimales

| Tipo | Descripción | Bytes ocupados | Literales | Rango de Valores |
|---|---|---|---|---|
| `float` | número decimal simple | 4 | 3.5f | simple presición 6 o 7 decimales |
| `double` | número decimal preciso | 8 | 0.123456789 | doble precisión 14 o 15 decimales |

- Otros tipos

| Tipo | Descripción | Bytes ocupados | Literales | Rango de Valores |
|---|---|---|---|---|
| `char` | número sin signo que representa el código de un caracter | dependiente de la codificación | 'A' | dependiente de la codificación |
| `boolean` | valor lógico | 1 | true o false | [true:false] |
| `String` | cadena de caracteres | dependeinte de la cantidad de caracteres de la cadena y la codificación | "Hola" | |

Ahora bien, conocemos algunos posibles tipos de datos pero cómo declaramos variables, simplemente anteponemos el tipo al nombre de la variable y con eso hemos declarado una variable y cubierto los tres vértices del triángulo que antes mencionamos tipo - nombre - memoria.

Por ejemplo algunas sentencias de declaración de variables:

```
int  x1;
float  x2, x3;
char c;
String nom;
```

> **Nota**: Un elemento a notar en las sentencias anteriores es que `String` no solo comienza con mayúscula sino que además no está marcada de color como `int`, `float` o `char`, la razón de esto es que String es una clase y como más adelante veremos en ese caso estamos declarando una referencia.

> Sin embargo, y debido al uso generalizado de las cadenas de caracteres en los lenguajes de programación String se vuelve necesario y por eso tiene un esquema especial de instanciación en Java.

Con las variables ya declaradas se pueden comenzar a usar asignando valores como hasta ahora lo hemos hecho en los demás lenguajes de programación.

Algunos ejemplos:

```
    // Declaración de variables
    char c;
    int a;
    String nom1, nom2;

    // Asignaciones
    c = '+';
    a = c;  // Notar la conversión, ya que como habíamos dicho un caracter es una
variable que almacen el código de la letra contenida.
    a = 'c'; // Igual al caso anterior.
    nom1 = "José";
    nom2 = n1; // Notar que estoy haciendo que nom2 tenga el mismo valor que nom1
    nom2 = "nom1"; // Aquí en cambio estoy dando a nom2 la cadena "nom1" y no el
valor de la variable

    // Notar que también se podría hacer ambas cosas en la misma sentencia
    int x = 10;
```

Nomenclatura de identificadores para variables, el nombre de las variables es elección de desarrollador pero debe cumplir ciertas restricciones y convenciones (en el Apunte 2 hicimos una aclaración de la diferencia de ambos conceptos), a continuación las principales restricciones y convenciones:

> - **Restricción**: El nombre o identificador de una variable en Java, solo puede contener letras (mayúsculas y/o minúsculas), o también dígitos (0 al 9), o también el guión bajo ( ` ) (también llamado guión de subrayado).
> - **Restricción**: El nombre de una variable no debe comenzar con un dígito.
> - **Restricción**: El nombre de una variable no puede ser una palabra reservada del lenguaje Java.
> - **Nota**: Java es case sensitive, es decir, hace diferencia entre minúsculas y mayúsculas, por lo que toma como diferentes a dos nombres de variables que no sean exactamente iguales. El identificador contador no es igual al identificador Contador y Java tomará a ambos como dos variables diferentes.
> - **Convención**: Java utiliza para los nombres de variable (como para casi todo), la notación camel case, es decir los identificadores se definen comenzando con minúscula siempre y se utiliza una sola letra mayúscula para indicar el comienzo de cada nueva palabra en el caso de un nombre compuesto.

## Inferencia de tipos en literales y conversión de tipos en java

Como dijimos Java es un lenguaje tipado y por lo tanto realiza un constante cheque de tipos de variables en cada momento de asignación u operación de valores en las variables mencionadas. Este chequeo se realiza a

tal punto que cuando asignamos un literal a una variable como por ejemplo:

```
    int a;
    a = 15;
```

Java antes de hacer la asignación del 15 en la variable a, está infiriendo el tipo del 15 (para java todo número entero es de tipo int) y luego al evaluar que la asignación es posible sin pérdida de precisión se realiza.

Pero en este otro caso:

```
    float b:
    // b = 3.5; // esta línea provocaría un error de compilación de posible
pérdida de precisión
    b = 3.5f
```

Java infiere el tipo de 3.5 como double ya que para java cualquier valor decimal es double, por ello tengo que intervenir y expresar que asumo que ese valor va a ser tratado como float agregando la letra f pegada al final del número.

Ahora bien, el problema se complica cuando queremos hacer asignaciones de valores de variables en otras variables, y aquí pueden ocurrir 3 casos diferentes. El primero y más feliz es que las variables a ambos lados del signo igual sean del mismo tipo, por ejemplo:

```
    int a, b;
    a = 10;
    b = a;
```

Java realiza el chequeo de tipos y entiende que no hay pérdida de precisión y por lo tanto realiza la asignación sin advertencia alguna. El segundo caso es que la variable a la izquierda del signo igual tenga mayor cantidad de memoria disponible que la variable a la derecha del signo igual, por ejemplo:

```
    short c = 15;
    int a;
    a = c;
```

> **Nota**: notar la situación contradictoria que se da en la asignación del 15, anteriormente dijimos que java asume los literales enteros como valores int y aquí estaríamos asignando un int en un short, sin embargo al ser un literal java puede hacer el control en tiempo de compilación y por lo tanto determina que la asignación no implica pérdida de precisión y permitirla.

Lo que en este caso sucede es que java detecta que no hay pérdida de precisión debido a que el short ocupa menos memoria que el int y por lo tanto, realiza una promoción implícita del valor de c a int y realiza la asignación con valores del mismo tipo.

Finalmente llegamos al caso crítico, donde el tamaño en memora de la variable ubicada a la izquierda del signo igual es **menor** que el tamaño de memoria de la variable ubicada a la derecha. En este caso se podría producir una pérdida de precisión porque podríamos tener un valor más grande en la variable de la derecha y java no podrá resolver qué poner en la variable de la izquierda, aquí el compilador Java nos va a requerir una acción explicita que se denomina **Casting** o conversión de tipo explicito, e implica anteponer el nombre del tipo al que quiero cambiar entre paréntesis al nombre de la variable a la que quiero cambiar el tipo en la asignación u operación. Por ejemplo:

```
int a = 123456;
short b;
// b = a; // esto provocaría error de compilación Possible lost of precision.
// Entonces tenemos que castear a la variable a
b = (short) a; // Esto sí compila
```

Sin embargo, si tenemos que adivinar ¿qué valor queda en la variable b?, lo primero que se nos podría ocurrir es que b queda valiendo el máximo para el tipo short: `32767`, pero si probamos el bloque anterior nos encontramos con que no es así, b queda valiendo: `-7616`. ¿Qué pasó aquí?

En realidad lo que pasó es que nosotros le dimos expresa instrucción a Java de que el valor de a debía ser entendido como entero, y eso hizo que java simplemente tome los dos bytes menos significativos de a y copie su contenido en b, con lo que copió los bits que estaban en esos bytes y esos bits conforman un número negativo.

## Caso particular, uso de var

La palabra reservada `var` es una característica introducida en Java 10 (LTS Java 11), que permite declarar variables locales de manera más concisa y flexible. Con var, el tipo de la variable se infiere automáticamente por el compilador en función del valor asignado a la variable.

Cuando utilizas la palabra clave `var` para declarar una variable, el compilador determina automáticamente el tipo de datos de esa variable basándose en el valor al que se le asigna. Esto significa que no es necesario especificar explícitamente el tipo de datos, lo que puede hacer que el código sea más claro y menos repetitivo. Por ejemplo

```
var edad = 25; // La variable "edad" es de tipo int
var nombre = "John"; // La variable "nombre" es de tipo String

System.out.println("Nombre: " + nombre + ", Edad: " + edad);
```

En este ejemplo, las variables `edad` y `nombre` son declaradas utilizando la palabra clave var. El compilador infiere automáticamente que `edad` es de tipo int debido a su valor numérico, y que `nombre` es de tipo String debido a su valor de texto.

var solo puede usarse para declarar variables locales dentro de métodos o bloques. No se puede usar para declarar parámetros de métodos, variables de instancia o variables de clase. **var no significa "variant"** (variable que puede cambiar de tipo). Una vez que se infiere un tipo para una variable con var, ese tipo es fijo. Se recomienda usar var con moderación y solo cuando el tipo de datos es obvio o cuando la inferencia de tipo mejora la legibilidad del código.

***Ventajas de var***:

- Reduce la repetición de tipos verbosos.
- Hace que el código sea más conciso y limpio.
- Facilita el mantenimiento del código al cambiar los tipos de variables sin necesidad de modificar las declaraciones de tipo.

***Desventajas de var***:

- Puede reducir la claridad si se usa de manera excesiva o en contextos donde el tipo no es obvio.
- Puede dificultar la comprensión del tipo de datos si el valor inicial es nulo o ambiguo.
- En resumen, la palabra clave var en Java permite la inferencia automática de tipos para variables locales. Ayuda a hacer el código más conciso y limpio, pero debe usarse con moderación y en situaciones donde el tipo sea obvio para mejorar la legibilidad.

## Operadores aritméticos en Java

Hemos visto que se puede asignar en una variable un valor, también se puede asignar el resultado de una expresión. Una expresión es una fórmula en la cual se usan operadores (como suma o resta) sobre diversas variables y constantes (que reciben el nombre de operandos de la expresión). Si el resultado de la expresión es un número, entonces la expresión se dice expresión aritmética. Los siguientes es un ejemplo de una expresión aritmética en Java:

```
int suma, num1 = 10, num2 = 7;
suma = num1 + num2;
```

En la última línea se está asignando en la variable suma el resultado de la expresión num1 + num2; y obviamente la variable suma quedará valiendo 17. Note que en una asignación primero se evalúa cualquier expresión que se encuentre a la derecha del signo =, y luego se asigna el resultado obtenido en la variable que esté a la izquierda del signo =. La siguiente tabla muestra los principales operadores aritméticos del lenguaje Java (volveremos más adelante con un estudio más detallado sobre la aplicación de estos operadores):

| Operador | Significado | Ejemplo de uso |
|---|---|---|
| + | suma | a = b + c; |

| Operador | Significado | Ejemplo de uso |
|---|---|---|
| - | resta | a = b - c; |
| * | producto | a = b * c; |
| / | división | a = b / c; |
| % | resto de una división | a = b % c; |

En Java los distintos operadores aritméticos actúan de acuerdo al tipo de las variables o constantes sobre las que operan. Así, si se usa el operador suma (+) para sumar dos variables int, el resultado será un valor int, y lo mismo ocurrirá con los demás.

Aunque lo anterior es lo lógico, no siempre el resultado obtenido será el que hubiésemos esperado: si se usa el operador división ( / ) y los dos números que se dividen son de tipo int, entonces el operador calculará la llamada división entera (o cociente entero) entre ambos, lo cual significa que la parte decimal del resultado será truncada. Veamos el siguiente ejemplo:

```
int a, b, c;
a = 5;
b = 2;
c = a / b;
```

El valor de c, será de 2, y no de 2.5, ya que la división se calculó en forma entera al ser de tipo int las variables a y b. Lo que java hace se denomina inferencia de tipo resultante de la operación y se basa en algunas reglas simples:

- Si los operadores son byte, short o int, el resultado será int aunque todos los operadores sean byte o short.
- A partir de int java asume el mayor tipo de los operadores involucrados según el siguiente orden:
  - `int < long < float < double`

En ese sentido, el operador resto ( % ) es muy útil porque permite calcular el resto de una división entera (y esto a su vez es muy valioso en casos en que se quiere aplicar conceptos de divisibilidad): la expresión r = x % y; calcula el resto de dividir en forma entera a x por y, y asigna ese resto en la variable r. En el caso citado antes, si los valores ingresados fueran x = 11 y y = 2, el resto calculado sería 1.

## Visualización por pantalla

Al ejecutar un programa, lo normal es que antes de finalizar el mismo muestre por pantalla los resultados obtenidos. En el lenguaje Java la instrucción más básica para hacer eso es `System.out.print()`. Esta instrucción permite mostrar en pantalla tanto el contenido de una variable como también mensajes formados por cadenas de caracteres, lo cual es útil para lograr salidas de pantalla “amigables” para quien use nuestros programas. La forma de usar la instrucción se muestra en los siguientes ejemplos:

```
public class App {
    public static void main(String[] args) {
        // declaramos e inicializamos variables
        int a, b;
        a = 3;
        b = a + 1;
        // mostramos el resultado por pantalla
        System.out.println(b);

    } // fin del main
} // fin del class
```

Aquí, la instrucción de la línea 12), muestra en pantalla el valor contenido en la variable b, o sea, el número 4. Sin embargo, una salida más elegante sería acompañar al valor en pantalla con mensaje aclaratorio: `System.out.print( “El resultado es: ” + b );`

Observar que ahora no sólo aparecerá el valor contenido en b, sino también la cadena “El resultado es: “, precediendo al valor. Notar también que para mostrar el valor de una variable, el nombre de la misma no lleva comillas, pues en ese caso se tomaría al nombre en forma literal. La instrucción que sigue, muestra literalmente la letra ‘b’ en pantalla: `System.out.print("b");`

## Entrada de Datos por teclado en Java

Esta operación permite mayor generalidad en la carga de datos de un programa. En el ejemplo anterior, se analizó un programa que permitía sumar dos números, en base al esquema de asignación directa de valores.

Podemos darnos cuenta rápidamente que un programa así planteado es muy poco útil, pues indefectiblemente el resultado mostrado en pantalla será 4... El programa tiene muy poca flexibilidad debido a que el valor inicial de la variable a es siempre 5 y el de b es siempre 3. Lo ideal sería que mientras el programa se ejecuta, pueda pedir que el usuario ingrese por teclado un valor para la variable a, luego otro para b, y que luego se haga la suma (en forma similar a como permite hacerlo una calculadora...)

Java a partir de la versión 5 provee una clase que está preparada para capturar una entrada por teclado en la consola estándar, de forma simple y sin necesidad de mayores conceptos previos. Java provee la herramienta que analizaremos a continuación como alternativa básica. Esta es la clase Scanner, la cual entre otras varias y amplias funcionalidades que implementa agrega también la posibilidad de realizar de manera simple y concreta la lectura de valores numéricos, caracteres o cadenas de caracteres desde teclado a través de la consola estándar.

Para utilizar la clase Scanner lo único especial que hay que agregar es la siguiente línea de código que debe quedar antes de la declaración de la clase en el archivo de código. `import java.util.scanner;`

Los métodos de la clase Scanner que nos interesan en este punto, los presentamos a continuación y cabe aclarar que la clase Scanner tiene varios métodos más, que no nos interesa analizar por ahora:

| Tipo de retorno | Método | Descripción |
|---|---|---|
| `String` | `nextLine()` | Retorna la próxima carga de cadena de caracteres. |

| Tipo de retorno | Método | Descripción |
|---|---|---|
| `boolean` | `nextBoolean()` | Retorna la próxima carga como un valor booleano. |
| `byte` | `nextByte()` | Retorna la próxima carga como un valor de tipo byte. |
| `double` | `nextDouble()` | Retorna la próxima carga como un valor de tipo double. |
| `float` | `nextFloat()` | Retorna la próxima carga como un valor de tipo float. |
| `int` | `nextInt()` | Retorna la próxima carga como un valor de tipo int. |
| `long` | `nextLong()` | Retorna la próxima carga como un valor de tipo long. |
| `short` | `nextShort()` | Retorna la próxima carga como un valor de tipo short. |

Para utilizar estos métodos es necesario realizar algunas tareas previas, en primer lugar, debemos crear el escáner, es decir debemos crear el objeto que va a escanear la entrada estándar para luego utilizar los métodos, ese bloque de código quedaría como sigue:

```
import java.util.Scanner;

public class App {
    public static void main(String[] args) {
        // Declarar la referencia a Scanner y crear la instancia que tome la
entrada estándar
        Scanner miEscaner = new Scanner(System.in);
        // Declara las variables a y b
        int a, b;

        System.out.print("Ingrese el valor de a: ");
        a = miEscaner.nextInt();
        System.out.print("Ingrese el valor de b: ");
        a = miEscaner.nextInt();

        // mostrar el resultado
        System.out.println("La suma es: " + (a+b));
    }
}
```

Cada uno de esos métodos se invoca escribiendo primero el nombre del objeto de la clase Scanner que creamos en la primera línea (en el ejemplo referenciado por `miEscaner`) seguido de un punto, y luego el nombre del método. Cada método, al ejecutarse, provoca que el programa entre en modo de espera, sin ejecutar ninguna otra instrucción hasta que alguien ingrese por teclado un valor y se presione la tecla `Enter`.

El valor así ingresado, se asigna en la variable indicada en la instrucción a la izquierda del signo igual, y luego prosigue normalmente el programa. En el ejemplo anterior, el segmento de programa mostrado provoca la carga por teclado de un número entero en la variable a, y otro en la variable b para hacer la suma pero ahora de los valores ingresados por teclado.

# Estructura alternativa o condicional

En problemas que no sean absolutamente triviales, es muy común que en algún punto se requiera comprobar el valor de alguna condición, y en función de ello proceder a dividir la lógica del algoritmo en dos o más ramas o caminos de ejecución. Por ejemplo, en un programa de control de acceso a un lugar seguro se debe pedir a cada usuario que cargue su clave de identificación. El programa entonces debería controlar si la clave cargada es correcta y sólo en ese caso habilitar el paso a esa persona. Pero si la clave fuese incorrecta, el programa debería tomar alguna medida alternativa, como sacar un mensaje de alerta por la consola de salida, bloquear una puerta, dar aviso a un supervisor, etc. Pero el hecho es que si sólo se emplean estructuras secuenciales de instrucciones, la situación anterior no podría resolverse.

Para casos así, los lenguajes de programación proveen instrucciones específicamente diseñadas para el chequeo de una o más condiciones, permitiendo que el programador indique con sencillez lo que debe hacer el programa en cada caso. Esas instrucciones se denominan estructuras condicionales, o bien, instrucciones condicionales.

En general, una instrucción condicional contiene una expresión lógica que puede ser evaluada por verdadera o por falsa, y dos bloques de instrucciones adicionales designados en general como la salida o rama verdadera y la salida o rama falsa. Si un programa alcanza una instrucción condicional y en ese momento la expresión lógica es verdadera, el programa ejecutará las instrucciones de la rama verdadera (y sólo esas). Pero si la expresión es falsa, el programa ejecutará las instrucciones de la rama falsa (y sólo esas).

En el lenguaje Java, una instrucción condicional típica como la que mostramos en la figura anterior, se escribe (esquemáticamente) así:

```
if (expresión lógica) {

    // instrucciones de la rama verdadera

}
else {

    // instrucciones de la rama falsa
}
```

La palabra reservada `if` da inicio a la estructura condicional que posee básicamente dos salidas o ramas: la rama o salida por verdadero y la rama o salida por falso. A continuación se escribe la condición que se evalúa, **siempre** entre paréntesis, y es la *expresión lógica* que se quiere evaluar por verdadero o falso, recordemos en este contexto, que una expresión lógica es una fórmula cuyo resultado es un valor lógico (o valor de verdad).

La "rama verdadera" se escribe entre llaves, inmediatamente después de cerrar el paréntesis de la condición, y la "rama falsa" va después de la rama verdadera, también envuelta entre llaves, pero precedida de la palabra reservada else. Si al evaluar la condición la misma es cierta, se ejecuta únicamente el bloque de instrucciones encerrado entre llaves de la rama verdadera, y se ignora la rama falsa. Si la condición fuera evaluada por falso, se ejecutará únicamente el bloque de la rama else, y será ignorada la rama verdadera.

## Expresiones lógicas, operadores relacionales y conectores lógicos en Java

En una lección anterior hemos visto que en general, una `expresión` es una fórmula compuesta por variables y constantes (llamados operandos) y por símbolos que indican la aplicación de una acción (llamados operadores). Hemos analizado también el uso de los llamados operadores aritméticos básicos de Java (suma, resta, producto, etc.) y sabemos que en función de esto, una expresión aritmética es una expresión cuyo resultado es un número.

Ahora bien, el hecho de que una expresión entregue como resultado un número, se debe a que los operadores que aparecen en ella son operadores aritméticos y por lo tanto llevan a la realización de alguna operación cuyo resultado será numérico. Sin embargo, en todo lenguaje existen operadores cuya acción no implica la obtención de un número como resultado, sino, por ejemplo, valores lógicos de la forma verdadero o falso (`true` o `false` en Java) y algunos otros operadores entregarán resultados de otros tipos (cadenas de caracteres, por ejemplo). En ese sentido, como ya hemos indicado, una expresión lógica es una expresión cuyo resultado esperado es un valor de verdad (`true` o `false`).

> **Nota**: en Java, a diferencia de otros lenguajes de programación, las sentencias que requieren un valor lógico, ***SOLO*** aceptan valores `boolean` es decir, no es válido utilizar valores de otro tipo para que el lenguaje los asuma verdaderos o falsos.

Como vimos, las instrucciones condicionales (y otros tipos de instrucciones que veremos, como las instrucciones repetitivas) se basan típicamente en chequear el valor de una expresión lógica para determinar el camino que seguirá el programa en su ejecución.

Para el planteo de expresiones lógicas, todo lenguaje de programación provee operadores que implican la obtención de un valor de verdad como resultado. Los más elementales son los llamados operadores relacionales u operadores de comparación, que en Java son los siguientes:

| Operador | Significado | Ejemplo | Observaciones |
|---|---|---|---|
| == | igual que | a == b | retorna `true` si `a` es igual que `b`, o `false` en caso contrario |
| != | distinto de | a != b | retorna `true` si `a` es distinto de `b`, o `false` en caso contrario |
| > | mayor que | a > b | retorna `true` si `a` es mayor que `b`, o `false` en caso contrario |
| < | menor que | a < b | retorna `true` si `a` es menor que `b`, o `false` en caso contrario |
| >= | mayor o igual que | a >= b | retorna `true` si `a` es mayor o igual que `b`, o `false` en caso contrario |
| <= | menor o igual que | a <= b | retorna `true` si `a` es menor o igual que `b`, o `false` en caso contrario |

Los operadores de la tabla anterior permiten plantear instrucciones condicionales en java para comparar de distintas formas dos valores.

También es posible emplear operadores conocidos como conectores lógicos para poder chequear varias expresiones lógicas a la vez. En general, cada una de las expresiones encadenadas por un conector lógico se designa como una proposición lógica. Por lo tanto, una proposición lógica es una expresión formada por variables y/o constantes relacionadas entre sí mediante operadores de comparación (o relacionales), de tal forma que el resultado de la expresión será un verdadero o un falso. Los tres principales conectores lógicos en Java se ven en la tabla siguiente

| Operador | Significado | Ejemplo | Observaciones |
|---|---|---|---|
| `&&` | conjunción lógica (y) | a == b && y != x | ver revisar tablas de verdad para estimar el resultado |
| `\|\|` | disyunción lógica (o) | n == 1 \|\| n == 2 | ver revisar tablas de verdad para estimar el resultado |
| `!` | negación lógica (no) | ! x > 7 | ver revisar tablas de verdad para estimar el resultado |

Un conector lógico u operador booleano es un operador que permite encadenar la comprobación de dos o más expresiones lógicas y obtener un resultado único. En general, cada una de las expresiones lógicas encadenadas por un conector lógico se designa como una proposición lógica. En la columna `Ejemplo` de la tabla anterior, las expresiones a == b, y != x, n == 1, n == 2 y x > 7 son proposiciones lógicas.

# Variantes de la expresión condicional en Java

## Condicional Doble

### Condicional Simple

También puede ocurrir (y de hecho es muy común) que para una condición sólo se especifique la realización de una acción si la respuesta es verdadera y no se requiera hacer nada en caso de responder por falso. Para estos casos, en Java y otros lenguajes es perfectamente válido escribir una instrucción condicional que sólo tenga la rama verdadera, omitiendo por completo la falsa. Una instrucción condicional de ese tipo se suele designar como condición simple, y en ella no se especifica la rama else: la instrucción condicional termina cuando termina la rama verdadera. La forma general típica de una instrucción condicional simple` en Java es la siguiente:

```
if (expresión lógica)
    //instrucciones de la rama verdadera
//continuación del programa
```

> **Nota**: si revisamos hay una sutil omisión en el fragmento anterior y es que no escribimos las llaves, eso es porque en el caso que el bloque de instrucciones a ejecutar incluya solo una instrucción las llaves se pueden omitir.
>
> En este caso puntual, el bloque condicional termina después de la primera sentencia de código. Esto aplica también a los condicionales tradicionales tanto para la rama verdadera como para la rama falsa.

Las instrucciones de la rama verdadera se encolumnan en un bloque hacia la derecha encerradas entre llaves si son más de una, del mismo modo que en un condicional tradicional, pero al terminar este bloque se escriben directamente las instrucciones para continuar con el programa, en la misma columna del `if` inicial, sin escribir la rama `else`.

### Condicional Múltiple

Como vimos en el ejemplo anterior, cuando es necesario evaluar un valor con más de dos elecciones posibles, se puede resolver con estructuras alternativas anidadas o en cascada, o si se quiere con estructuras simples en secuencia. Sin embargo, si se tiene un problema donde el número de alternativas es grande, usar estos métodos, puede plantear serios problemas de escritura del algoritmo y naturalmente de legibilidad. Es por eso que existe una estructura llamada selección, decisión o condicional múltiple que considera estos casos en particular.

La estructura de decisión múltiple evaluará una expresión que podrá tomar n valores distintos: 1, 2, 3,..., n. Según que elija uno de estos valores en la condición, se realizará una de las n acciones, o lo que es igual, el flujo del algoritmo seguirá un determinado camino entre los n posibles.

En Java, la sintaxis para esta estructura es la siguiente:

```
    switch (expresión_entera) {

        case (valor1) :
            // instrucciones para valor 1
            break;

        case (valor2) :
            // instrucciones para valor 2
            break;

        ...

        case (valorN) :
            // instrucciones para valor N
            break;


        default:
            // instrucciones si no cumple ninguna de las anteriores

    }
```

> *Nota*: este es el caso tradicional de switch, a partir de la versión 7 de java también se pueden usar cadenas de caracteres lo que, si bien agrega overhead por la comparación de cadenas, hace extremadamente más legibles los fragmentos de código.
>
> Además, a partir de la próxima versión LTS (Java 21), también se incluirá la posibilidad de utilizar expresiones regulares en las ramas del switch

Creo sin embargo, que una de las particularidades clave a destacar aquí es que la estructura switch en java funciona en cascada, esto es, si la expresión coincide con `valor2` pero yo omito intencionalmente codificar los break, entonces java ejecutará todas las sentencias de la rama `case (valor2)` y todas las sentencias de las demás ramas hasta encontrar un break o el final de la estructura.

**Operador ternario o condicional inline**

El operador ternario, también conocido como operador condicional inline, es una estructura que permite tomar decisiones basadas en una condición y asignar valores diferentes a una variable en función de si la condición es verdadera o falsa. A diferencia de las estructuras if-else, el operador ternario es más conciso y se utiliza para expresar lógica condicional de manera más compacta.

El operador ternario es útil y debería ser utilizado siempre que tengamos condicionales que en cada una de sus ramas asignan a una misma variable un valor diferente. Por ejmplo:

```
int edad = 18;
String mensaje;

if (edad >= 18) {
    mensaje = "Mayor de edad";
}
else {
    mensaje = "Menor de edad";
}
```

En estos casos como veremos el operador ternario logra un fragmento de código más claro y legible. El operador ternario se compone de tres partes: la condición, el valor si la condición es verdadera y el valor si la condición es falsa. La sintaxis general es:

`variable = (condición) ? valor_verdadero : valor_falso;`

```
int edad = 18;
String mensaje = (edad >= 18) ? "Mayor de edad" : "Menor de edad";
```

En este ejemplo, se utiliza el operador ternario para asignar el valor de la variable status basado en la edad. Si la edad es mayor o igual a 18, la variable status tendrá el valor "Mayor de edad"; de lo contrario, tendrá el valor "Menor de edad". Y, como podemos ver, este fragmento cuando nos acostumbramos es mucho más legible y flexible que la versión original.

***Ventajas del Operador Ternario***:

- Concisión: El operador ternario permite expresar lógica condicional en una sola línea de código, lo que puede hacer que el código sea más limpio y conciso.
- Legibilidad: En algunos casos, el operador ternario puede hacer que la lógica condicional sea más legible que una estructura if-else.

***Desventajas del Operador Ternario***:

- Complejidad Limitada: El operador ternario es útil para decisiones simples, pero puede volverse menos legible y más complejo cuando se manejan decisiones más complicadas.

- Dificultad en el Mantenimiento: Un uso excesivo o mal considerado del operador ternario puede dificultar el mantenimiento del código y la comprensión de otros desarrolladores.

***Consideraciones al Usar el Operador Ternario***:

- Utiliza el operador ternario cuando la lógica condicional sea simple y fácilmente comprensible.
- No abuses del operador ternario para evitar comprometer la legibilidad del código.
- Si la lógica condicional es compleja, considera usar una estructura if-else en su lugar.
- En resumen, el operador ternario en Java es una forma concisa de expresar lógica condicional en una sola línea de código. Sin embargo, debe usarse con moderación y solo en casos donde la lógica sea simple y fácil de entender.

## Operadores resumidos en Java

A medida que se avanza en el estudio y planteo de algoritmos y programas para problemas cada vez más complejos, se verá que en la mayoría de esos programas será necesario eventualmente llevar a cabo procesos de `conteo` (por ejemplo, determinar cuántas veces apareció un número negativo), o de sumarización (por ejemplo, determinar cuánto vale la suma de todos los valores que tomó la variable `x` a lo largo de la ejecución de programa, suponiendo que `x` cambia de valor durante esa ejecución). El primer caso se resuelve incorporando una variable de conteo (o simplemente un contador), y el segundo, incorporando una variable de acumulación (o simplemente un acumulador).

En ambas situaciones se trata de variables que en una expresión de asignación aparecen en ambos miembros: la misma variable se usa para hacer un cálculo y para recibir la asignación del resultado de ese cálculo. Los siguientes son dos ejemplos de expresiones de conteo o acumulación (en el primero, la variable `a` se usa como un contador, y en el segundo la variable `b` se usa como un acumulador):

```
a = a + 1

b = b + x
```

En general, un contador es una variable que sirve para contar ciertos eventos que ocurren durante la ejecución de un programa. Intuitivamente, se trata de una variable a la cual se le suma el valor 1 cada vez que se ejecuta la expresión. Esto es así porque contar normalmente significa sumar 1.

Entonces, técnicamente, un contador es una variable que actualiza su valor en términos de su propio valor anterior y de 1.

La sentencia resumida para expresar la expresión de conteo en java, es similar a otros lenguajes como python y puede expresarse así: `a += 1` la cual funciona así:

- En primer lugar se ejecuta la parte derecha de la asignación, con el valor actual de `a`. Si a comenzó valiendo cero, entonces la primera vez que se ejecute la expresión `a + 1` se obtiene un uno.
- En segundo lugar se asigna el valor así obtenido en la misma variable `a`, con lo cual se cambia el valor original.

Del mismo modo que en la expresión `a = a + 1` la variable `a` funciona como contador, en forma similar la variable `c` en la expresión `c = c - 1` funciona como decrementador: va restando de a uno a partir del valor original de la variable `c`. En los siguientes ejemplos mostramos contadores de formas diversas (asegúrese de entender lo que cada instrucción hace cada vez que se ejecuta):

```
a = a + 1
b = b – 1
c = c + 2
d = d * 4
e = e / 3
```

Por otra parte, un acumulador o variable de acumulación es básicamente una variable que permite sumar los valores que va asumiendo otra variable o bien otra expresión en un proceso cualquiera. Técnicamente, y en general, un acumulador es una variable que actualiza su valor en términos de su propio valor anterior y el valor de otra variable u otra expresión.

Según la definición dada de un acumulador, las siguientes expresiones también son expresiones de acumulación:

```
s = s + x
b = b \* z
a = a – y
p = p / t
c = c + 2\*x
```

Es interesante notar que en Java (como en otros lenguajes) cualquier expresión de conteo o de acumulación_responde a la forma general siguiente:

`variable = variable operador expresión`

donde `variable` es la variable cuyo valor se actualiza (y aparece en ambos miembros de la expresión de asignación) y `expresión` es una constante, una variable o una expresión propiamente dicha (formada a su vez por constantes, variables y operadores). Y el hecho es que en el lenguaje Java cualquier expresión que venga escrita en la forma general anterior, se puede escribir también en la forma resumida siguiente:

`variable operador= expresión`

A modo de ejemplo, veamos las siguientes equivalencias:

| Forma general | Forma resumida |
|---|---|
| a = a + 1 | a += 1 |
| b = b – 1 | b –= 1 |
| c = c + 2 | c += 2 |

| Forma general | Forma resumida |
|---|---|
| d = d * 3 | d *= 3 |
| e = e / 4 | e /= 4 |
| s = s + x | s += x |
| b = b * z | b *= z |
| a = a – x | a –= x |
| p = p / t | p /= t |
| c = c + 2*x | c += 2*x |

## Operadores unarios, pre y post incremento / decremento

Siguiendo el esquema anterior pero yendo más allá aún, tenemos dos casos particulares que se dan para el contador, es decir el incremento de una variable en terminos de su propio valor y 1 más, o el decremento de una varaible en términos de su propio valor y 1 menos, según lo visto en esos casos tendríamos:

```
a += 1
b -= 1
```

Para estos casos casos particulares java agrega 4 operadores extra, en realidad dos operadores que pueden ser utilizados en dos formas distintas cada uno. A saber:

| Forma resumida | Operador unario |
|---|---|
| a += 1 | a++ o ++a |
| b -= 1 | b-- o --b |

Estoas dos formas son denominadas pre y post incremento para el operador `++` y pre y post decremento para el operador `--`, la primera pregunta que surge es: ¿qué sentido tiene que existan dos? y la primera respuesta que surge es que más allá de alguna discusión muy fina que por ahí leí, si están como una sentencia independiente no tienen diferencia alguna.

Ahora la cosa cambia cuando la expresión de incremento está a la derecha del signo igual o como parámetro de un método. En esos casos el resultado puede no ser tan claro:

| Valores iniciales | Sentencia | Valores finales |
|---|---|---|
| a = 5 | c = ++a | a => 6 ; C => 6 |
| a = 5 | c = a++ | a => 6 ; C => 5 |
| b = 5 | c = --b | b => 4 ; C => 4 |
| b = 5 | c = b-- | b => 4 ; C => 5 |

La posición del operador define el orden de ejecución de la sentencia, si es pre incremento o pre decremento, primero se va a llevar la modificación de la variable incrementada y luego la asignación y en caso de ser post incremento o post decremento primero se va a llevar a cabo la asignación y luego la modificación de la variable incrementada.

## Estructura repetitiva

Existen dos tipos generales de ciclos, los cuales se designan con los siguientes nombres genéricos (el motivo de estos nombres se verá oportunamente en esta misma lección):

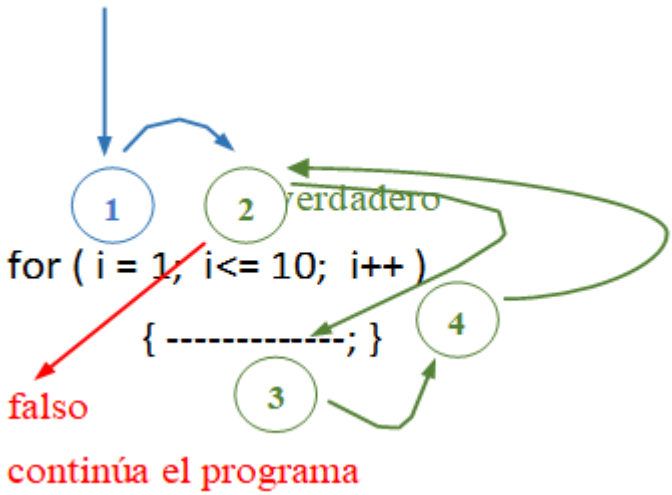
- Ciclos "0 – N"
- Ciclos "1 – N"

Estos ciclos son implementados por los diversos lenguajes en formas que varían ligeramente de un lenguaje a otro. El lenguaje Java implementa los dos tipos de ciclos mediante tres instrucciones específicas, designándolos respectivamente con las palabras reservadas que se marcan a continuación en letra negrita:

- Ciclo **for** ( ciclo del tipo "0-N" )
- Ciclo **while** ( ciclo del tipo "0-N" )
- Ciclo **do while** ( ciclo del tipo "1-N" )

### El ciclo for en Java

El ciclo for comúnmente se usa en circunstancias en las cuales se conoce de antemano la cantidad exacta de repeticiones que deben realizarse, aunque en el lenguaje Java la sintaxis del ciclo es tan amplia que admite ser usado en cualquier circunstancia que requiera repetición de acciones. De hecho, veremos que un ciclo for es una variante de un ciclo while_. Un ejemplo con la estructura de un for se ve en el modelo siguiente, suponiendo que se desea repetir 10 veces la ejecución de ciertas instrucciones (otros casos requieren variantes directas sobre la estructura mostrada):




En la cabecera del ciclo existen tres secciones: la primera (`i=1` en este caso) se llama sección de inicialización, y se usa para indicar el valor inicial de las variables de control del ciclo. La segunda (`i<=10` en este caso) es la condición de control del ciclo: si la condición es verdadera el ciclo ejecuta una repetición del bloque de acciones, pero si es falsa corta el ciclo y continúa el programa. La tercera sección (`i++` en este caso) es la

sección de incremento, en la cual se indica la forma en que cambiará el valor de cada variable de control del ciclo.

Cuando el ciclo comienza, se ejecuta primero (y por única vez) la sección de inicialización, dando un valor inicial a la variable de control (ver acción (1) en el gráfico). Luego, se verifica la condición de control (2). Si la misma da "verdadero", se ejecuta entonces el bloque de acciones del ciclo (3), y al finalizar, se produce un retorno automático a la cabecera, específicamente a la sección de incremento, para cambiar el valor de las variables de control (4). Finalmente, se vuelve a verificar la condición de control. Si vuelve a dar "verdadero", se repite lo anterior, y si da "falso", el ciclo se detiene.

Observar que si la condición de control fuera falsa la primera vez que se evalúa, entonces el bloque de acciones no sería ejecutado. Esta es la característica básica de los ciclos tipo "0-N": el cero hace referencia a la cantidad mínima de veces que se espera que el bloque del ciclo sea ejecutado, y la N es un valor genérico que se refiere a que una vez comenzado el ciclo, se espera que el bloque de acciones se ejecute N veces, siendo N un valor indefinido pero que se supone será conocido al momento de ejecutar el ciclo.

A modo de ejemplo el siguiente fragmento imprime por pantalla los números del 1 al 5 entre llaves y separados por comas:

```
System.out.print('{');
for (int i = 1; i <= 5; i++) {
    System.out.print(i);
    if (i < 5)
        System.out.print(',');

}
System.out.println('}');
```

La salida sería:

```
{1, 2, 3, 4, 5}
```

## El ciclo while en Java

En muchas ocasiones necesitamos plantear un ciclo que ejecute en forma repetida un bloque de acciones pero sin conocer previamente la cantidad de vueltas a realizar. Para estos casos la mayoría de los lenguajes de programación, y en particular Java, proveen un ciclo designado como ciclo `while`.

Como todo ciclo, un ciclo while está formado por una cabecera y un bloque o cuerpo de acciones, y trabaja en forma general de una manera muy simple. La cabecera del ciclo contiene una expresión lógica que es evaluada en la misma forma en que lo hace una instrucción condicional if, pero con la diferencia que el ciclo while ejecuta su bloque de acciones en forma repetida siempre que la expresión lógica arroje un valor verdadero. Así como un if hace una única evaluación de la expresión lógica para saber si es verdadera o falsa, un ciclo while realiza múltiples evaluaciones: cada vez que termina de ejecutar el bloque de acciones vuelve a

evaluar la expresión lógica y si nuevamente obtiene un valor verdadero repite la ejecución del bloque y así continúa hasta que se obtenga un falso.

La característica principal del ciclo while es que la condición de control se evalúa por primera vez antes de la primera ejecución del bloque de acciones. Al igual que en el ciclo for_, esto implica que si en la primera evaluación de la condición de control se obtiene un valor falso, entonces el bloque del ciclo no será ejecutado y por esta causa el ciclo es del tipo "*0-N*".

Por ejemplo, el siguiente fragmento implementa un ciclo que se ejecuta mientras el valor de la variable cont se mantenga menor o igual a 5. En cada vuelta, se muestra un mensaje que contiene el número de vuelta en que se encuentra el ciclo:

```
int cont = 0
while (cont <= 5) {
    cont++;
    System.out.println("Vuelta número: " + cont);
}
```

La salida por consola estándar si se ejecuta este programa, sería la siguiente:

```
Vuelta número: 1
Vuelta número: 2
Vuelta número: 3
Vuelta número: 4
Vuelta número: 5
```

La variable cont_recibió un valor inicial (el cero en este caso) antes de comenzar la ejecución del ciclo (de no ser así, la evaluación de la condición la primera vez no tendría sentido...). El bloque del ciclo contiene varias instrucciones y figura encerrado entre llaves. Cuando el programa llega al ciclo por primera vez, evalúa la condición. En este caso, el valor inicial de cont es cero y la condición es cierta: cont es menor que 5. Siendo cierta la condición, se ejecuta el bloque de acciones, con lo cual cont suma 1 y luego se muestra el mensaje: Vuelta número: 1. Termina allí el bloque de acciones, pero como se trata de un ciclo, automáticamente el programa regresa a verificar otra vez la condición: si el nuevo valor de cont sigue siendo menor que 5, el bloque se ejecuta otra vez y así seguirá hasta que en algún retorno la condición sea falsa. Cuando cont tenga el valor 5, semostrará el mensaje Vuelta número: 5 y el ciclo se detendrá. En ese momento se ejecutarán las instrucciones que se hayan escrito debajo del ciclo.

## El ciclo do while en Java

La característica básica del ciclo do while es que la condición de control se evalúa por primera vez después de ejecutar por primera vez el bloque de acciones. Esto implica que al menos una vez el bloque de acciones del ciclo siempre será ejecutado, independientemente del valor inicial de la condición de corte (por eso este ciclo es del tipo "*1-N*": una vez como mínimo se ejecuta el bloque, y luego puede llegar a N repeticiones). La estructura de un ciclo do while es la siguiente:

El esquema de funcionamiento del ciclo do while es el siguiente: cuando se llega a una línea que comienza con do_, se ejecuta el bloque de acciones que sigue a continuación, sin importar el valor de la condición de control. Luego de ejecutar ese bloque, se evalúa la condición de control. Si dicha condición arroja un falso, el ciclo corta y continúa el programa con la instrucción que esté debajo del ciclo. Si la condición se valúa en verdadero, se provoca un retorno automático a la línea marcada con do_, y se vuelve a ejecutar el bloque de acciones del ciclo. El proceso continuará de esta forma, hasta obtener un falso en la condición de control.

Un ejemplo general de ciclo `do ... while`

```
do {
    // acciones a repetir
} while(cond);
```

### Elementos de control de ciclo Java

El bloque de acciones de un ciclo (*while o for*) en Java puede incluir una instrucción `break` para cortar el ciclo de inmediato sin retornar a la cabecera para evaluar la expresión lógica de control o incluir la sentencia `continue` para volver desde ese punto a la cabecera sin terminar el bloque de sentencias repetitivas.

El siguiente segmento tiene el objetivo mostrar el uso de `break` y `continue`:

```
System.out.print('{');
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0)
        continue;
    System.out.print(i);
    if (i == 7)
        break;
    if (i < 10)
        System.out.print(',');

}
System.out.println('}');
```

Ahora es un poco menos intuitiva la salida esperada:

```
{1, 3, 5, 7}
```

## Uso de Scanner para leer un archivo de texto

Como vimos en el apartado anterior la clase Scanner es capaz de configurarse con la entrada estándar de java para tomar lo que el usuario carga por teclado sin embargo el uso más común de la clase Scanner es el de la lectura de archivos o flujos de texto.

Es decir nosotros podemos tomar un archivo de texto com por ejemplo el archivo llamado datos.txt con el siguiente contenido:

```
1
2
3
4
5
```

Y procesarlo con a partir de una instancia de la clase Scanner de la siguiente manera:

```
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.io.File;


public class App
{
    public static void main( String[] args ) throws FileNotFoundException
    {
        File f = new File("datos.txt");
        Scanner miEscaner = new Scanner(f);

        while (miEscaner.hasNext()) {
            System.out.println(miEscaner.nextInt());
        }

    }
}
```

Con la salida que efectivamente pensamos que obtendríamos de manera intuitiva:

```
1
2
3
4
5
```

## Novedades de Java moderno: `switch ->` y bloques de texto `"""`

Desde Java 14 en adelante (y consolidado en Java 21), Java ha incorporado mejoras que simplifican la escritura y lectura del código, sin perder tipado fuerte ni claridad.

### Switch mejorado con `->`

La sintaxis tradicional de `switch` puede ser verbosa y propensa a errores si se omiten los `break`. A partir de Java 14 se puede usar una forma más compacta y segura:

```
String dia = "LUNES";

switch (dia) {
    case "LUNES", "MIERCOLES", "VIERNES" -> System.out.println("Día de cursado");
    case "SABADO", "DOMINGO" -> System.out.println("Fin de semana");
    default -> System.out.println("Día normal");
}
```

> 💡 No es necesario escribir `break` y se pueden agrupar múltiples valores en una sola rama con comas.

## 📄 Bloques de texto multilínea con `"""`

Otra novedad muy úctil es el uso de *text blocks* para definir cadenas de texto multilínea de forma clara:

```
String html = """
    <html>
        <body>
            <h1>Bienvenidos</h1>
        </body>
    </html>
    """;

System.out.println(html);
```

> 📝 Este formato es ideal para definir código HTML, JSON, mensajes, o cadenas largas sin concatenaciones ni `\n`.

## ☑️ ¿Por qué las usamos?

| Característica | Desde versión | Ventaja didáctica |
|---|---|---|
| `switch ->` | Java 14 | Evita errores con `break`, más legible |
| Bloques `"""` | Java 15 | Facilita trabajar con texto multilínea |

> En esta materia vamos a usar estas características modernas porque hacen el código más legible, más seguro y más conciso. ¡Aprovechamos que Java 21 es LTS y lo permite!