

Polito - Curso de programación 2025

Documentación y ejercicios <3

UB Polito Arcuschin

© Para que reine en el pueblo el amor y la igualdad 🙌

Índice

1. Bienvenidxs :)	5
2. Clase 00: Historia y fundamentos	6
2.1 Clase 0: Historia de la programación y fundamentos.	6
3. Clase 01: Programa y Operadores	7
3.1 ¿Qué es un programa? Fases de desarrollo estructura y sintaxis	7
3.1.1 Breve repaso	7
3.1.2 Programa	8
3.1.3 Fases de desarrollo	8
3.1.4 Estructura de un programa	9
3.1.5 Tipos de datos y valores	10
3.1.6 Operadores	13
3.2 Ejercicios para la clase 01	16
3.2.1 Ejercicio 1 🧠	16
3.2.2 Ejercicio 2	16
3.2.3 Ejercicio 3	16
4. Clase 02: Variables y Expresiones	18
4.1 Clase 02: Variables + Expresiones	18
4.1.1 Variables	18
4.1.2 Constantes	20
4.1.3 Expresiones	21
4.1.4 Palabras reservadas	22
4.2 Ejercicios para la clase 02	23
4.2.1 Ejercicio 01: Suma de enteros	23
4.2.2 Ejercicio 02: Área de un rectángulo	23
4.2.3 Ejercicio 03: Promedio de tres números	23
4.2.4 Ejercicio 04: Intercambio de valores	24
4.2.5 Ejercicio 05: Conversión de temperatura	24
4.2.6 Ejercicio 06: Cálculo de sueldo	24
4.2.7 Ejercicio 07: Edad en meses y días	24
4.2.8 Ejercicio 08: Concatenación de cadenas	24
4.2.9 Ejercicio 09: Número par o impar	24
4.2.10 Ejercicio 10: Hipotenusa (Teorema de Pitágoras)	24
5. Clase 03: Variables II	25
5.1 Clase 03: Más variables	25
5.1.1 Conversión de tipos de datos (casteo)	25

5.1.2	Clase Scanner	26
5.2	Ejercicios para la clase 03	28
5.2.1	<code>java.util.Scanner</code>	28
5.2.2	Ejercicio 11: Área de un círculo	28
5.2.3	Ejercicio 12: Perímetro de un cuadrado	28
5.2.4	Ejercicio 13: Velocidad promedio	28
5.2.5	Ejercicio 14: Minutos a horas y minutos	28
5.2.6	Ejercicio 15: Dígito de las unidades	28
5.2.7	Ejercicio 16: División entera y resto	28
5.2.8	Ejercicio 17: Potencia	28
5.2.9	Ejercicio 18: Kilogramos a libras	28
5.2.10	Ejercicio 19: Promedio de notas con decimales	28
5.2.11	Ejercicio 20: Doble y triple de un número	28
5.2.12	Ejercicio 21: Área y perímetro de un triángulo equilátero	29
5.2.13	Ejercicio 22: Conversión de metros a cm y mm	29
5.2.14	Ejercicio 23: Promedio de velocidad en metros por segundo	29
5.2.15	Ejercicio 24: Descuento en un producto	29
5.2.16	Ejercicio 25: Tiempo de viaje	29
5.2.17	Ejercicio 26: Concatenación de edad	29
5.2.18	Ejercicio 27: Interés simple	29
5.2.19	Ejercicio 28: Interés compuesto	29
5.2.20	Ejercicio 29: Conversión de dólares a pesos	29
5.2.21	Ejercicio 30: Conversión de segundos a horas:minutos:segundos	29
5.2.22	Ejercicio 31: Promedio ponderado	29
5.2.23	Ejercicio 32: Promedio de edad de un grupo	29
5.2.24	Ejercicio 33: Separar decenas y unidades	29
6.	Clase 04: Arrays	30
6.1	Clase 04: Arrays (vectores y matrices)	30
6.1.1	Definición	30
6.1.2	Vectores (Arrays unidimensionales)	30
6.1.3	Matrices (Arrays bidimensionales)	32
6.1.4	Matrices (Arrays tridimensionales)	34
6.1.5	En resumen	38
6.2	Ejercicios para la clase 04	40
6.2.1	Ejercicio 01	40
7.	Clase 05: Estructuras de control	41
7.1	Estructuras de control de flujo	41
7.1.1	Estructuras condicionales	42

7.1.2	Operador de negación !	43
7.2	Ejercicios para la clase 05	44
7.2.1	Ejercicio 01	44
8.	Clase 06: Funciones II	45
8.1	Clase 06: Funciones segunda parte	45
8.2	Ejercicios para la clase 06	46
8.2.1	Ejercicio 01	46
9.	Clase 07: API de Java	47
9.1	Clase 07: API de Java	47
9.2	Ejercicios para la clase 07	48
9.2.1	Ejercicio 01	48
10.	Clase 08: Clases	49
10.1	Clase 08: Clases	49
10.2	Ejercicios para la clase 08	50
10.2.1	Ejercicio 01	50
11.	Clase 09: Arrays	51
11.1	Clase 09: Arrays (o vectores)	51
11.2	Ejercicios para la clase 09	52
11.2.1	Ejercicio 01	52
12.	Glosario	53
13.	Ejercicios	54
13.1	Ejercicios sueltos	54

1. Bienvenidxs :)

En este lugar iremos cargando las clases, ejercicios y otros recursos que servirán como la documentación del curso de programación de la UB Polito Arcushin del año 2025.

La idea es que puedan recurrir a estas páginas para refrescar conceptos entre clases y tener ejercicios a mano para practicar.

Como ya hemos dicho muchas veces: todos es perfectible, así que no duden en buscar errores 🐞 o cosas que no quedan claras 😕 y avisarnos para que podamos revisar los contenidos.

¡Esperamos que aprendan cosas nuevas, se les ocurran ideas extravagantes y sobre todo que disfruten la experiencia de pensar con otrxs!

Abel, Mariana, Cami y Manu 🙋

2. Clase 00: Historia y fundamentos

2.1 Clase 0: Historia de la programación y fundamentos.

3. Clase 01: Programa y Operadores

3.1 ¿Qué es un programa? | Fases de desarrollo | estructura y sintaxis

3.1.1 Breve repaso

En la clase pasada habíamos llegado a una definición para la acción de programar. Vimos que programar no era exclusivo de la informática, sino una forma de pensar (lógica) para elaborar una serie de instrucciones (algoritmo) que serán ejecutadas por una máquina o sistema.

? ¿Qué es programar?

Programar es una forma de pensar para elaborar una serie de instrucciones que serán ejecutadas por un sistema.

? ¿Qué es un algoritmo?

- Un algoritmo es una creación humana.
- Se lo puede definir como una serie de instrucciones ordenadas o una lista ordenada de pasos para lograr un objetivo.
- Ejemplos clásicos: Una receta de cocina 📖🍲, un manual para armar un mueble 📄🔧, tomarse el colectivo 🚏

Vimos también qué era un lenguaje de programación: una forma de escribir esas instrucciones. Y los categorizamos por algunas de sus posibles características: por su forma de ejecución, por su nivel de rigurosidad en su sintaxis y semántica, por sus paradigmas y por su nivel de abstracción.

Vimos también que los lenguajes de programación de "alto nivel" intentan arrimarse al lenguaje humano y, al igual que ese, tienen una serie de reglas de sintaxis y semántica con las que hay que cumplir para que, al fin de cuentas, la máquina pueda ejecutar nuestras instrucciones de la manera en que las pensamos.

✎ Los lenguajes de programación

- Un lenguaje de programación es una forma de escribir instrucciones.
- Se los suele agrupar por sus características distintivas.
- Al igual que el lenguaje humano, cada lenguaje tiene reglas **sintácticas** y **semánticas**.

A partir de estas definiciones, ya podemos adentrarnos en dilucidar... ¿qué es entonces un programa?

3.1.2 Programa

La palabra programa puede hacer referencia a muchas cosas no tan disímiles como aparentan: Un programa de una materia de la facu, un programa de televisión, un programa político (quienes tengan cercanía a espacios de discusión política, habrán notado los constantes reclamos de la existencia de dicha cosa), etc...

Como podrán observar, todas esas acepciones de la palabra "programa" son bastante similares: Es un conjunto ordenado de tópicos que se deben seguir en el orden establecido y de una forma más o menos constante.

Pues bien, entonces un programa informático no es más que una seguidilla de sentencias que se ejecutan desde el inicio hasta el final (ya veremos un poco más esto), realizando así las acciones que definimos dentro de ese programa.

? ¿Qué es un programa?

Un programa informático es un conjunto de sentencias que se ejecutan desde el inicio hasta el final realizando así las acciones definidas.

3.1.3 Fases de desarrollo

Esto lo veremos en detalle más adelante cuando veamos ejercicios de *problemas*, pero vale la pena mencionarlo aquí para tener una idea general de las fases que tiene el desarrollo de un programa.

¿Por qué insistimos tanto en esto de que programar no es exclusivamente escribir código en un lenguaje elegido? La programación empieza mucho antes de sentarse a escribir: **se arranca siempre pensando un problema**, antes incluso de pensar en cómo resolverlo. Recuerden que dijimos en la primera clase que programar es pensar en soluciones creativas para problemas complejos.

Es decir, primero vamos a intentar **descomponer el problema en pasos lógicos** y ordenados, luego vamos a **imaginar y diseñar soluciones posibles** y, finalmente como último paso, vamos a **traducir esas soluciones a código** en el lenguaje que hayamos elegido.

☰ Fases de desarrollo

1. Descomponer el problema en pasos lógicos y ordenados
2. Imaginar y diseñar las soluciones posibles
3. Traducir las soluciones elegidas a código

3.1.4 Estructura de un programa

Vamos a ver cómo se ve un programita básico ("Hola Mundo"¹) para intentar descomponer su estructura.

```

1  package holamundo;
2
3  /**
4   *
5   * @author nombre
6   */
7
8  public class HolaMundo {
9
10     /**
11      * @param args the command line arguments
12      */
13     public static void main(String[] args) {
14         // TODO: el código a ejecutar viene acá abajo.
15         System.out.print("¡Hola mundo!");
16     }
17 }

```

Descomponemos el programa:

- Vemos que hay **líneas** de texto
- Las líneas parecieran estar agrupadas en "bloques"
- Hay llaves que "engloban" líneas
- Hay paréntesis que "abrazan" palabras
- Hay unas líneas "especiales" que empiezan con símbolos: barras y asteriscos (`/**` `/**`)
- La línea que imprime el texto (`System.out.print()`), "ya viene" con el lenguaje.

Sentencias y bloques

Las líneas de nuestro programa son... ¡las sentencias! Una sentencia es una representación de la acción que nuestro programa debe ejecutar.

A la agrupación de líneas de código que conforman una sola sentencia, las llamaremos bloques.

Llaves y paréntesis

Las llaves son las que construyen la estructura del código, delimitando los bloques. Indican que todo lo que está entre la llave de apertura (`{`) y la de clausura (`}`) "pertence" a la línea que antecede inmediatamente a la apertura.

En nuestro ejemplo tenemos dos "conjuntos" de llaves:

```

public class HolaMundo {
    // Hay cosas acá adentro.
    // Y todas pertenecen a esta clase (class) pública (public) llamada HolaMundo.
}

```

```

public static void main(String[] args) {
    // Hay cosas acá adentro.
    // Y todas pertenecen esta función pública y estática (static) llamada main.
}

```

Los paréntesis también cumplen la función de agrupar, pero a diferencia de las llaves no conforman bloques, sino algo similar a un agrupamiento de palabras o incluso una palabra sola.

Veremos esto más en detalle cuando veamos "Funciones" (que se suele representar: $f(x)$) pero por ahora recordemos que lo que está dentro de un paréntesis "pertenece" a la *palabra* o expresión que antecede inmediatamente el de apertura (.

```

1  /**
2   *
3   * @author nombre
4   */
5
6  public class HolaMundo {
7
8      /**
9       * @param args the command line arguments
10      */
11     public static void main(String[] args) {
12         // TODO: el código a ejecutar viene acá abajo.
13         System.out.print("¡Hola mundo!");
14     }
15 }

```

Indentación

El espacio en blanco que vemos antes de las líneas de código es la indentación o el indentado. En la sintaxis de los lenguajes humanos esto se conoce como sangría. En el caso de Java la indentación no tiene importancia sintáctica ni semántica, pееееero es importante respetarla porque facilita visualmente la comprensión y la identificación de los bloques.

Comentarios

Un comentario es una línea de texto que no es ejecutada, es una nota que agrega quien escribe el programa para hacer aclaraciones sobre ese código. Pueden ser notas sobre la funcionalidad (qué hace), o para facilitar la comprensión de la solución implementada (cómo lo hace).

Es una buena práctica utilizar los comentarios para "documentar" nuestro código y así facilitar que otras personas puedan modificarlo (o incluso para nuestro *yo del futuro* :)).

Ciclo del programa: Inicio, proceso, final.

Veremos esto en clase cuando "corramos" 🏃 nuestro programa, pero dejamos aquí una definición muy esquemática de un ciclo.

Analicemos el ciclo:

- **Inicio:** Punto de entrada (o *entrypoint* en inglés)
- **Proceso:** Una vez que se cargó en memoria, el código es ejecutado y el sistema operativo gestionará los recursos del entorno (asignando recursos, comunicando con dispositivos de entrada y salida, archivos, etc...).
- **Final:** Código de salida.

3.1.5 Tipos de datos y valores

Los tipos de datos **clasifican datos según sus características específicas**. Los valores son los datos en sí mismos (y pertenecerán a un tipo u otro). Entenderemos un poco más el concepto de valor y dato cuando veamos Variables.

Recuerdan que en la primera clase vimos que había lenguajes fuertemente tipados o débilmente tipados, esta característica de los lenguajes hará que en nuestro programa debamos explicitar (o no) el tipo de dato que vamos a manipular. Veamos entonces ahora los tipos de datos llamados primitivos o elementales que nos ofrece Java.

Tipos primitivos

Los "tipos primitivos"², a veces llamados también Tipos Elementales, son los tipos de datos originales de cada lenguaje. Es decir que cada lenguaje ya nos proporciona esos tipos y así también los define:

1. Qué **tipo de dato** se puede representar (números -enteros o decimales-, caracteres -letras y símbolos-, etc...)
2. Sus **rangos de valores posibles**
3. Y por lo anterior entonces también se define **el espacio que ocuparán en memoria** los datos pertenecientes a cada tipo

Recordemos que las máquinas solamente "entienden" datos que llamamos **binarios**, esto quiere decir que sólo pueden ejecutar unos y ceros. Por lo tanto, para las máquinas todo será una combinación de esos dos valores a los que definimos como **bits**: Un bit puede tener un valor de 1 o de 0.

Esto lo tenemos que tener claro para entender como se definen los tipos de datos primitivos.

Aprovechemos el primer tipo de datos para profundizar en esto:

BYTE (BYTE - ANGLICISMO- U OCTETO)

Es un tipo de dato que representa una estructura de **8 bits**. O sea, es el conjunto de 8 bits:

```
byte = 00000000
```

Este tipo de dato puede almacenar **valores numéricos enteros** que van desde -128 hasta 127 (ambos inclusive).

Ustedes se preguntarán ¿cómo es que eso sucede? bueno, la máquina construye los números alternando **1** y **0** en esa sucesión de bits.

Por ejemplo, lo que escribimos más arriba: `byte = 00000000` para la máquina representa **0**. O sea, nosotros escribimos un **0** en el teclado y la máquina lo "traducirá" a ese conjunto de (8) bits. Si nosotros escribimos un 5, la máquina lo traducirá como `00000101`. Como pueden ver siguen siendo 8 bits, lo que cambian son las posiciones que tienen un **1** o un **0**.

Todo esto es importante porque les da la dimensión de lo que ocupan en memoria los datos.

Cuando nosotros creamos un dato, la computadora reserva en memoria el tamaño mayor que puede llegar a alcanzar ese dato. En el ejemplo que estamos usando, un byte, la máquina va a reservar 8 bits (8 lugares para poner unos y ceros).

Entonces es importante que **antes de crear datos pensemos qué valores vamos a querer manejar**.

SHORT (ENTERO CORTO)

Representa un tipo de dato de **16 bits**. O sea, que usará el doble de memoria que nuestro tipo anterior: `00000000 00000000`.

Puede almacenar **valores numéricos enteros desde -32.768 hasta 32.767**. Por supuesto también lo hace combinando **1**s y **0**s en esas 16 posiciones.

INT (ENTERO)

Este tipo de dato puede almacenar **valores numéricos enteros** de 32 bits, o sea **4 veces nuestro primer dato** `00000000`. Los valores van desde -2^{31} hasta $2^{31}-1$. Como pueden ver, el tamaño del número que permite manejar es mucho más grande.

Podemos trabajar con números un poco superiores a los 2 mill millones -2.147.483.648 y 2.147.483.647.

LONG (ENTERO LARGO)

Es un tipo de datos de **64 bits** o sea **8 veces nuestro primer tipo de dato**: `00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000`. Y puede almacenar **números enteros que van desde -2^{63} hasta $2^{63}-1$** . Como se puede imaginar es extremadamente grande: 9.223372×10^{18}

FLOAT (COMA FLOTANTE O 'REAL')

Es un tipo de dato de **32 bits**, como el int, pero permite manejar números con coma flotante. Por ejemplo `16,4` o `1879223,45`. Estos números son los que usualmente se conocen como "decimales".

Este tipo de dato tiene una precisión simple, esto es algo que veremos más adelante, pero por el momento alcanza con saber que no debe ser utilizado si vamos a necesitar números con alta precisión (es decir, lo podemos usar para aplicaciones gráficas o videojuegos, pero no para información científica o financiera).

DOUBLE (COMA FLOTANTE DOBLE O 'REAL LARGO')

En este caso es un tipo de dato de **64 bits**, como el long, pero permite manejar números con coma flotante, como el float.

Las diferencias entre el float y double son:

- El rango numérico (es el doble)
- La precisión (es, también, el doble) Al ser un número decimal su "exactitud" depende de la cantidad de dígitos que puede manejar después de la coma. Por eso, el tipo de dato "double" tiene el doble de precisión que el "float".

BOOLEAN (BOOLEANO -ANGLICISMO- O LÓGICO)

Este es un tipo de dato de **un bit** que representa verdadero (en inglés "true") o falso (en inglés "false") (que también podría ser sí o no, 1 o 0, lleno o vacío, etc...).

Se los llama así porque está basado en el álgebra "booleana", del filósofo y matemático inglés [George Boole](#).

CHAR (CARÁCTER O SÍMBOLO)


Es un tipo de dato que representa un carácter [Unicode](#) de **16 bits**. Se utiliza mucho el tipo de dato char para manejar un símbolo o una letra (¡que es en última instancia un símbolo!) y no un fragmento de texto.

STRING (CADENA DE TEXTO)

Este tipo de dato lo ponemos como primitivo ¡pero no lo es! Lo incluimos aquí porque es tan utilizado como los primitivos.

En principio hay que observar que empieza con una mayúscula. En casi todos los lenguajes de programación orientada a objetos esto tiene un significado importante. Por ahora podemos decir que cuando tenemos que trabajar con textos (frases o palabras), usamos este tipo de dato.

Tipos primitivos - Tabla

 Tipos primitivos de JAVA				
Tipo	Traducción	Memoria utilizada	Rango de valores	Breve descripción
byte	Byte u Octeto (raro)	1 byte (8 bits)	de -128 a 127	Representación de un número entero positivo o negativo.
short	Entero corto	2 byte (16 bits)	de -32.768 a 32.767	Representación de un entero positivo o negativo con rango corto.
int	Entero	4 byte (32 bits)	de -2^{31} a $2^{31}-1$	Representación de un entero estándar.
long	Entero largo	8 byte (64 bits)	de -2^{63} a $2^{63}-1$	Representación de un entero de rango ampliado.
float	Coma flotante o Real	4 byte (32 bits)	de -10^{32} a 10^{32}	Representación de un número real estándar. La precisión del dato contenido varía en función del tamaño del número: la precisión se amplía con números más próximos a 0 y disminuye cuanto más se aleja del mismo.
double	Real largo	8 byte (64 bits)	de -10^{300} a 10^{300}	Representación de un número real con mucha precisión.
char	Carácter	2 byte (16 bits)	de '\u0000' a '\uffff'	Carácter o símbolo. Recordar que para utilizar una "cadena de texto" se debe usar la clase String.
boolean	Booleano o Lógico	1 bit	true - false	Representa una 'posición' lógica con dos valores posibles: verdadero (true) o falso (false)

3.1.6 Operadores

Los operadores son símbolos o palabras que utilizamos para manipular y combinar expresiones. Son elemento que nos permiten realizar, como su nombre lo indica, operaciones como:

- Comparaciones
- Procedimientos lógicos
- Cálculos aritméticos
- Asignaciones

Operadores relacionales

Utilizamos los operadores relacionales cuando necesitamos comparar dos elementos o valores.

Dependiendo el operador que usemos tendremos que tener en cuenta, o no, la posición de los elementos con relación al operador.

OPERADOR	DESCRIPCIÓN	EJEMPLO
<code>==</code>	Es igual	<code>a == b</code>
<code>!=</code>	Es distinto	<code>a != b</code>
<code><</code>	Menor que	<code>a < b</code>
<code>></code>	Mayor que	<code>a > b</code>
<code><=</code>	Menor o igual que	<code>a <= b</code>
<code>>=</code>	Mayor o igual que	<code>a >= b</code>

Es decir, decir que `a == b` es lo mismo que decir que `b == a`, sin embargo, no sería lo mismo intercambiar los operandos cuando hacemos una comparación o relación mayor/menor: `a < b` no es lo mismo que `b < a`.

Operadores lógicos

A diferencia de los relacionales que **comparan** elementos o valores, los operadores lógicos **combinan** operaciones *booleanas*.

Son muy utilizados en estructuras condicionales y *loops*, que veremos más adelante, para combinar condiciones y obtener luego un resultado.

OPERADOR	DESCRIPCIÓN	EJEMPLO	EJEMPLO COTIDIANO
<code>&&</code>	Y (de adición)	si [condición] Y [otra condición]	Agarro la campera si hace frío Y llueve
<code> </code>	O (de exclusión)	si [condición] U [otra condición]	Agarro la campera si hace frío O llueve
<code>!</code>	NO (de negación)	si NO [condición]	Dejo la campera si NO hace frío

Cuando usamos los operadores `&&` si nuestra primera condición devuelve *false* o usando el comparador `||` sin nuestra primera condición devuelve *verdadero*, en ambos casos la segunda condición no se evaluará.

Los operadores de negación simplemente invierten un valor lógico.

Operadores aritméticos

Los operadores aritméticos se utilizan para realizar operaciones matemáticas básicas.

OPERADOR	DESCRIPCIÓN
<code>+</code>	Adición
<code>-</code>	Resta
<code>*</code>	Multipliación
<code>/</code>	División
<code>%</code>	Módulo

Operadores de asignación

Los operadores de asignación se utilizan para asignar valores a las variables.

OPERADOR	DESCRIPCIÓN
=	Asignación simple
+=	Añadir y asignar
-=	Restar y asignar
*=	Multiplica y asigna
/=	Divide y asigna
%=	Módulo y asignar

-
1. "Hola, mundo" en informática es un programa que muestra el texto «Hola, mundo» en un dispositivo de visualización, en la mayoría de los casos la pantalla de un monitor. Este programa suele ser usado como introducción al estudio de un lenguaje de programación, siendo un primer ejercicio típico considerado fundamental desde el punto de vista didáctico. ←
 2. En todos los elementos de Java vamos a ver que estos reciben nombres en inglés. Junto a cada tipo vamos a poner su traducción en castellano para entenderlos, pero cuando los utilizemos siempre tendrá que ser con su nombre inglés. ←

3.2 Ejercicios para la clase 01

En esta clase vimos cómo crear nuestro primer programa (HolaMundo) y como hacer para que nos imprima algo en la consola del IDE.

Primeros pasos para estos ejercicios:

1. Abrir la IDE (Netbeans)
2. Crear un nuevo programa (o abrir uno existente)
 - a. Si estás creando uno nuevo, acórdete de seleccionar la opción "Java with Ant".

3.2.1 Ejercicio 1 🍷

Usando las herramientas del sistema (`System.out.print`), impriman un texto en la consola de Netbeans.

Y ahora jueguen usando los operadores. Por ejemplo, podemos sumar dos cadenas de texto.

```
System.out.print("Hola," + " " + "¿qué tal?");
```

Podemos también sumar o "juntar" caracteres:

```
// No me acuerdo ni uno, pero https://www.unicode.org/charts/script/chart_Symbol-Other.html
System.out.println("H" + '\u2665' + "1" + "u");

System.out.println("Venceremos " + '\u270C');

System.out.println("Extraño la " + '\u00F1');
```

3.2.2 Ejercicio 2

Utilicen el otro método para imprimir mensajes, para que agregue un salto de línea cuando termina el texto (`System.out.println`).

Esto nos va a venir bien para que no nos quede todos los mensajes ahí amontonados en una sola línea :)

¿Qué otro operador se puede usar con cadenas de texto? ¿Podremos dividir, restar o multiplicar?

Hagan la prueba, cuando Java no sepa qué hacer con el operador y la cadena, se los va a decir en forma de error cuando compilen.

Sugerencias:

- Prueben usando operadores relacionales.

3.2.3 Ejercicio 3

Usando este último método, prueben imprimir mensajes de texto y prueben también si pueden hacer cuentas y comparar valores.

Recuerden que si queremos imprimir texto usamos las comillas envolviendo el texto, pero los números y operadores deben ir por fuera de las comillas.

Ponemos acá ejemplos para que se guíen, pero ¡no los copien tal cual! Inventen cualquiera cosa, sin miedo, que la mejor manera de aprender es insistir en el error, tratar de entenderlo y después arreglarlo para llegar al resultado esperado :)

Unos ejemplos básicos para arrancar:

```
// Multiplicamos y comparamos.
System.out.println(2*3 == 6);

System.out.println(2*3 == 2 + 2 + 2);

// Comparamos dos cadenas.
System.out.println("peras" == "manzanas");
```



```
// Atenti al uso de los paréntesis.
System.out.println("¿las peras NO son manzanas? " + ("peras" != "manzanas"));
```

```
// Multiplicamos.
System.out.println(2*3);
```

Prueben ahora agregarle una cadena de texto (string):

```
// Agregamos texto a la cuenta.
System.out.print("Se sabe que 2 x 3 es: " + 2*3);
//
//Método pa' imprimir Cadena de texto OP Cuentita
```

Bueno, parece que todo marcha bien, probemos qué pasa si en lugar de multiplicar sumamos...

```
// Agregamos texto a la cuenta.
System.out.print("Todo el mundo sabe que habrá un 5: " + 2 + 3);
```

!?! ¿Qué pasoooó?! 😞

Les dejamos este interrogante para que traten de entender por qué no sumó, sino que "juntó" los números.



Pistas:

- Cuando multiplicamos, todo salió como esperamos. ¿Cuál puede ser la diferencia?
- ¿Se acuerdan de matemática y las precedencias (o el "orden de evaluación")? Algo muy parecido está pasando acá.

Un pequeño adelanto: Esta herramienta que usamos se descompone en:

- `System` → Es una clase (`class`) de la librería Java. De allí su nombre "sistema".
- `out` → Es un atributo (o campo) de la clase. Esta palabra es "salida" en español. Bastante clarito su nombre :)
- `print` y `println` son dos métodos de la clase. `Print` es "imprimir" y `println` probablemente sea la contracción de "imprimir línea". Es decir que termina la línea escribiendo una línea nueva (lo que nosotrxs hacemos cuando apretamos la tecla enter).

4. Clase 02: Variables y Expresiones

4.1 Clase 02: Variables + Expresiones

En esta clase veremos qué son las variables y constantes, aprenderemos a definir las y usarlas junto con los elementos que vimos hasta ahora (nuestros tipos primitivos, cadenas de texto o `String` y operadores). Veremos también a qué se llama "expresión" y cómo las construimos.

4.1.1 Variables




Una variable es un contenedor en el que se guarda un valor o dato. Sería algo similar a una caja donde *metemos* un valor y a la que le ponemos una etiqueta en el frente. La etiqueta es el nombre o, técnicamente hablando, el **identificador** de la variable, que nos va a permitir acceder a ella (y así a su valor), manipularla y reutilizarla.

Las variables nos permiten reutilizar valores tantas veces como queramos sin tener que estar repitiendo ese valor cada vez. El nombre de la variable se convierte entonces en una **representación del valor** que le *asignamos* (atención a esta palabrita).

Por otro lado, y como lo indica su nombre, las variables pueden cambiar su valor a lo largo de nuestro programa. Por ejemplo, una variable `nombre` puede tener un valor inicial de "**Juan Domingo**" y luego ese valor puede transformarse en "**Cristina**".

Variable

Una variable es un elemento en el que podemos almacenar datos.

- Tienen un identificador (o nombre) 
- Su valor puede modificarse a lo largo de la ejecución del programa 
- En Java, debemos especificar su *tipo* 

¿Cómo creamos entonces una variable? Bien sencillo, vamos a escribir el nombre o **identificador** de nuestra variable, pero ¡OJO! ** En Java, al ser un lenguaje de tipado estricto, debemos además decirle al sistema qué tipo de dato vamos a querer guardar ahí. Esto se llama en programación **declaración de variables**. Porque al crearla estamos también declarando su **tipo** y comprometiéndonos con el sistema a sólo guardar valores que sean admitidos por ese tipo. Veamos:

Imaginemos que estamos trabajando en un sistema que manejará la compra de entradas en una sala de teatro, y necesitamos entonces en algún momento de nuestro programa definir y guardar la cantidad total de asientos que tiene la sala.

```
byte totalAsientos;
```

Con esa línea ya estamos declarando nuestra variable y le estamos diciendo al sistema que `totalAsientos` tendrá un valor del tipo `byte`. El sistema entonces nos tomará la palabra y reservará en memoria ese cachito de espacio minúsculo que requiere un byte. Noten como nuestra línea termina con un punto y coma. Eso forma parte de la sintaxis de Java y lo veremos más adelante cuando veamos [Expresiones](#).

Vamos ahora a **asignarle** un valor a nuestra flamante variable. ¿Les suena de algún lado esa palabra "asignar"? Ciertamente, ya la habíamos visto cuando vimos [Operadores](#), así que usaremos el operador de asignación (`=`) para asignarle un valor:

```
byte totalAsientos; // --> Aquí la declaramos.

totalAsientos = 125; // --> Aquí le asignamos un valor.
```

Veán como cuando la declaramos, debemos definir su tipo, pero luego cuando queremos asignarle un valor, simplemente ponemos su identificador seguido del operador de asignación y el valor.

Estas dos líneas de código también se pueden sintetizar en una sola si queremos **declarar** nuestra variable Y **asignarle** un valor en el mismo momento, simplemente unimos las dos sentencias:

```
byte totalAsientos = 125;
//   ^           ^   ^   ^
// TIPO     NOMBRE  OP  VAL
```

De esta manera estaremos declarando nuestra variable de tipo byte y le asignaremos un valor numérico de 125.

Usando nuestro primer ejemplo con cadenas podemos ver como es que se le cambia el valor a una variable:

```
String nombre = "Juan Domingo";
System.out.println("Nombre inicial: " + nombre);
// Acá pasan cosas.
System.out.println("Pasan muchos años y muchas cosas...");
nombre = "Cristina";
System.out.println("Nombre luego: " + nombre);
```

! Atención máxima 🙄

Ahora bien, ¿qué pasa si un tiempo después nos llaman de la sala de teatro y nos dicen que remodelaron la sala y la ampliaron, que ahora tiene 300 butacas? Bueno, en principio si nos caían bien nos vamos a poner contentxs, y luego buscaremos nuestro código e iremos a modificar nuestra variable que tenía el total de asientos para actualizarla con el nuevo valor. Pareciera tan sencillo como encontrar la línea donde la declaramos y simplemente cambiar el número:

```
byte totalAsientos = 300;
```

Prueben declarar esta variable de este modo ^ a ver qué sucede 🌟. Por más que la diferencia sea muy chiquita 🤏 para Java ese no es un valor válido.

Cuando decimos que el valor de la variable puede cambiar y el sistema no nos va a reprochar nada, pues las *variables varían*... debemos prestar mucha atención a un tema ya a esta altura recurrente: **Java es un lenguaje estricto con sus tipos de datos**.

Ya nos habíamos comprometido con el sistema a guardar valores de tipo byte en nuestra variable `totalAsientos`, y el sistema ya nos había reservado ese espacio en memoria, pero resulta que `300` ¡no es un valor válido en el tipo `byte`! (1)

1. Rango de valores válidos de byte: [-128 .. 127].

Así que si intentamos cambiar el valor inicial de nuestra variable a 300 lo que sucederá es que el sistema simplemente no nos va a dejar y nos indicará el error ¹.

Por eso es importante siempre intentar anticiparse a este tipo de cambios o, dicho de otra forma, no confiarse en que las especificaciones no cambiarán. Siempre será mejor trabajar con tipos de datos que tengan un buen margen para esos cambios.

En este ejemplo que dimos queda muy evidente que nos estamos "pasando" porque simplemente le asignamos un valor más alto cuando declaramos la variable, pero imaginen el siguiente ejemplo en donde una variable `totalEntradas` no tiene un valor definido por nosotrxs 'a mano', sino que hay una serie de operaciones que definen su valor. Por ejemplo:

```
byte totalEntradas;

// Unas cosas muy interesantes que hace nuestro programa.

byte entradasVendidas = estoVieneDeUnCalculoComplejo;
byte invitaciones = estoVieneDeOtroCalculo;

totalEntradas = entradasVendidas + invitaciones;
```

Si la operación de suma que determina el valor de nuestra variable es un número entero más grande que 127, vamos a estar en problemas.

Para situaciones muy específicas en donde necesitamos explicitar que una variable no puede cambiar su valor, es decir, para declarar variables que no varían (¡qué contradicción!) podemos hacer uso de otro elemento. Esos elementos se llaman "constantes".

4.1.2 Constantes

Las constantes son un tipo especial de variable. Y su nombre ya nos dice qué es lo que será diferente en este elemento: su *mutabilidad*. Evidentemente, las constantes son... constantes y no pueden variar su valor a lo largo del programa.

Constantes

Las constantes son un contenedor en el que podemos guardar datos o valores.

- Tienen un identificador 📌
- Su valor es inalterable durante toda la ejecución del programa 🔒
- En Java, debemos especificar su *tipo* 🏠

Cualquier dato que no cambie es buen candidato para guardarse en una constante. Ejemplos:

- El número pi (π)
- La cantidad de meses que tiene un año
- La velocidad de la luz
- La burguesía Argentina :P

La declaración de las constantes es igual a la de las variables, pero le tenemos que sumar una palabra que las definirá como constantes: `final`. Veamos unos ejemplos.

```
// Declaramos las constantes porque hay cosas que nunca cambian (¿?).
final double NUMERO_PI = 3.1415926535;
final String MEJOR_PAIS = "Argentina";
```

Identificadores

Para usar los identificadores, que no son más que el 'nombre' que le damos a nuestras variables y constantes, debemos tener algunas reglas en cuenta.

Sintaxis de los identificadores

- No se pueden utilizar **palabras reservadas**
- El identificador debe ser único.
- Deben comenzar siempre por una letra (1).
- Los siguientes caracteres pueden ser letras, dígitos, guión bajo (`_`) o el signo de pesos (`$`).
- Se distingue entre mayúsculas y minúsculas (2).
- No hay una largo máximo ni mínimo establecido.

1. La gente de Oracle (la empresa que desarrolló Java) recomienda NO utilizar guiones bajos (`_`) ni signo de pesos (`$`) como primer carácter de un identificador aunque estos estén permitidos.
2. **A** Java es sensible a mayúsculas y minúsculas, con lo cual, el identificador `coso` será diferente a `Coso` o `COSO` (o cualquier otra variación).

Además de estas reglas de sintaxis hay otro conjunto de cuestiones a tener en cuenta que se suelen denominar "buenas prácticas" de programación.

Estas son, como su nombre lo indica, una serie de buenos modales a la hora de escribir código, que pueden variar de lenguaje en lenguaje. No van a hacer que nuestro código falle en sentido estricto, pero seguirlos le facilita (¡mucho!) la lectura de nuestro código tanto a otras personas como a nosotros mismos.

Hay bastantes estándares de programación que se pueden seguir y, por supuesto, siempre hay competencia entre cuál estándar es mejor, pero con el tiempo uno suele adoptar el que sea más utilizado en el lenguaje o tecnología que desarrolla o incluso en el ámbito social en el que programamos (grupo de amigos, trabajo, etc...).

Dentro de las buenas prácticas más extendidas de Java, hay algunas que aplican a los identificadores y las detallamos a continuación:

✓ Buenas prácticas

- Usar identificadores que reflejen el **significado** o el **uso** de los elementos. Ejemplos:
 - `unasCosas` ✗
 - `cantidadSillas` ✓
- Usar "**camelCase**" (es un anglicismo que se podría traducir como "tipoCamello") para identificar variables. Ejemplos:
 - `cantidad_sillas` ✗
 - `cantidadesillas` ✗
 - `cantidadSillas` ✓
- Usar mayúsculas para identificar constantes. Ejemplos:
 - `mejor_pais` ✗
 - `MejorPais` ✗
 - `MEJOR_PAIS` ✓
- Se recomienda no usar nombres muy largos ni muy cortos. Los nombres cortos solo se recomiendan para variables *temporales* de corta vida útil.
- No podremos usar tildes, diéresis ni ñ en nuestros identificadores 💔
 - `añoActual` ✗
 - `canciónFinal` ✗
 - `PINGÜINOS` ✗

4.1.3 Expresiones

Junto con los operadores, las **expresiones son los elementos básicos con los que armaremos nuestras sentencias**.

Repasemos qué era una sentencia: visualmente las vimos como líneas de código (1) que representan una **acción** que debe ser ejecutada. Los bloques nos permiten agrupar líneas de código que serán "leídas" como una única sentencia.

1. En Java -y otros lenguajes- terminan con un punto y coma `;`

En términos muy muy generales diremos que **una sentencia NO devuelve nada** y que **una expresión SÍ devuelve alguna cosa**.

Podemos definir una expresión como un fragmento de código que utiliza operadores para manipular valores y producir un resultado.

Lo que en matemática llamamos "hacer una cuenta" sería en programación "crear una expresión". La diferencia principal es que las expresiones en programación además de valores numéricos (como en las cuentas) podemos también trabajar con palabras, caracteres y *booleanos*.



Expresiones

Una expresión es un conjunto de operadores, valores, variables, constantes y funciones que devuelve un resultado.

Veamos un ejemplo básico para determinar qué es una expresión, qué es un operador y qué es una sentencia y poder así diferenciarlos.

```
c = a + b;
```

- `a`, `b` y `c` : Son **valores** o datos.
- `=` y `+` : Son **operadores**.
- `a + b` : Es una **expresión**.
- `c = a + b;` : Es una **sentencia**.

4.1.4 Palabras reservadas



Palabras reservadas en Java

<code>abstract</code>	<code>continue</code>	<code>for</code>	<code>new</code>	<code>switch</code>
<code>assert</code>	<code>default</code>	<code>goto</code>	<code>package</code>	<code>synchronize</code>
<code>boolean</code>	<code>do</code>	<code>if</code>	<code>private</code>	<code>this</code>
<code>break</code>	<code>double</code>	<code>implement</code>	<code>protected</code>	<code>throw</code>
<code>byte</code>	<code>else</code>	<code>import</code>	<code>public</code>	<code>throws</code>
<code>case</code>	<code>enum</code>	<code>instanceof</code>	<code>return</code>	<code>transient</code>
<code>catch</code>	<code>extends</code>	<code>int</code>	<code>short</code>	<code>try</code>
<code>char</code>	<code>final</code>	<code>interface</code>	<code>static</code>	<code>void</code>
<code>class</code>	<code>finally</code>	<code>long</code>	<code>strictfp</code>	<code>volatile</code>
<code>const</code>	<code>float</code>	<code>native</code>	<code>super</code>	<code>while</code>

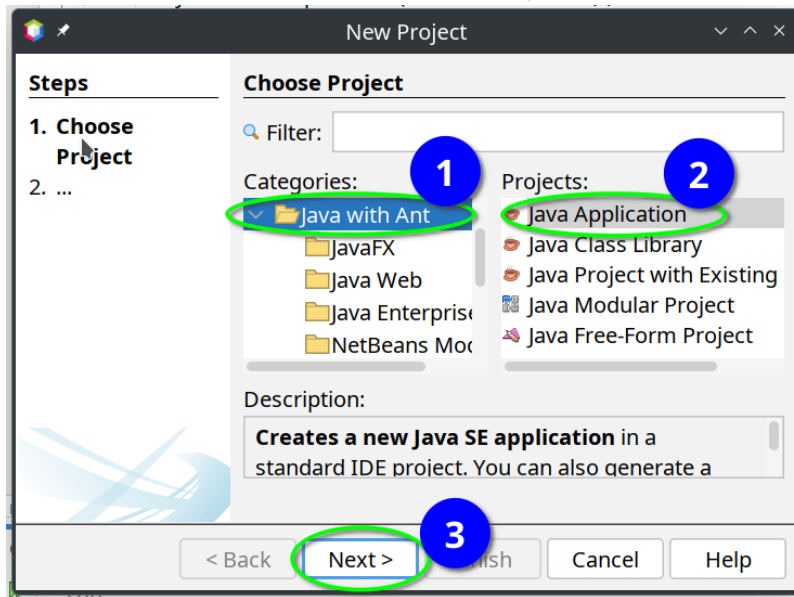
¹. En situaciones excepcionales no nos va a tirar un error, sino que obtendremos valores totalmente inesperados. Ya veremos esto en detalle más adelante. ←

4.2 Ejercicios para la clase 02

En esta clase vimos variables, constantes y expresiones.

Primeros pasos para estos ejercicios:

- Abrir la IDE (Netbeans)
- Crear un nuevo programa (o abrir uno existente)
- Si estás creando uno nuevo, acordate de seleccionar la opción "Java with Ant".



NOTA: Para todos estos ejercicios, cuando proponemos mostrar 'cosas' por consola, agreguen además un pequeño texto descriptivo antes del resultado o valor, así se entiende qué es lo que estamos imprimiendo 🖨️ ❤️

Ejemplo:

```
int desde = 2003;
int hasta = 2015;
int laDecadaGanada = hasta - desde;

System.out.println("La década ganada duró: " + laDecadaGanada + " años.");
```

4.2.1 Ejercicio 01: Suma de enteros

Declará dos variables enteras `a` y `b`, asignales valores y mostrá por consola la suma de ambas.

4.2.2 Ejercicio 02: Área de un rectángulo

Escribí un programa que calcule el área de un rectángulo, utilizando la fórmula `base * altura`. Declará una variable para la base y otra para la altura, ambas de tipo `double`. Finalmente, mostrá el resultado en la consola.

4.2.3 Ejercicio 03: Promedio de tres números

Imaginemos que tenemos que sacar el promedio de una materia que tuvo 3 exámenes en el cuatrimestre.

Escribí un programa en el que:

1. Cada nota sea una variable de tipo **entera** (`int`)
2. Se calcule el promedio de las 3 notas
3. El promedio se guarde en otra variable que debemos mostrar por consola, **con decimales** (OJO: ¿qué tipo debemos usar para mostrar decimales?)

4.2.4 Ejercicio 04: Intercambio de valores

Escribí un programa en el que inicialmente tengas dos variables (`x = 5` e `y = 10`) y luego intercambiá sus valores y mostrá el resultado por consola.

4.2.5 Ejercicio 05: Conversión de temperatura

Imaginemos que tenemos un tío paquete que vive en Nueva York y cada vez que hablamos por teléfono nos pregunta cómo está el clima en Buenos Aires. Sabiendo que en Argentina utilizamos la escala de temperatura en grados Celsius (°C) y que en Estados Unidos utilizan la escala Fahrenheit (°F), escribamos un programita que nos ayude a convertir la temperatura de Celsius a Fahrenheit para ahorrarnos la engorrosa tarea de hacer esta operación manual cada vez que hablamos con él.

1. Declará una variable para guardar la temperatura actual de Buenos Aires en grados Celsius.
2. Usá la fórmula $F = (C * 9/5) + 32$ para convertir escalas.
3. Guardá la temperatura en F en una variable y
4. Mostrá la temperatura en F por consola

4.2.6 Ejercicio 06: Cálculo de sueldo

Guardá en variables el sueldo base y un bono extra. Calculá el sueldo total y mostralo. (Pensá si los sueldos y el bono son valores enteros o reales)

4.2.7 Ejercicio 07: Edad en meses y días

Porque nunca está de más ser extravagante, escribí un programa que muestre tu edad en meses y en días (aproximados, usando 365 días para un año, -a los bisiestos los ignoramos por un momentito).

Mostrá ambos resultados en la consola.

4.2.8 Ejercicio 08: Concatenación de cadenas

Utilizá una variable para guardar tu primer nombre y otra para guardar tu apellido. Mostrá por consola un mensaje que diga `Hola, [nombre] [apellido]`.

4.2.9 Ejercicio 09: Número par o impar

Utilizá una variable de tipo entera para guardar un número, mostrá en consola si el número es par o impar.

4.2.10 Ejercicio 10: Hipotenusa (Teorema de Pitágoras)

1. Utilizá dos variables `catetoA` y `catetoB` (`double`) para guardar los valores de los catetos de un triángulo,
2. calculá la hipotenusa con la fórmula: $h = \sqrt{a^2 + b^2}$
3. y mostrá el resultado por consola

5. Clase 03: Variables II

5.1 Clase 03: Más variables

5.1.1 Conversión de tipos de datos (casteo)

En programación es indispensable la conversión de datos. Esto se debe a que en algunos casos por más que evaluemos y estudiemos los datos que necesitamos para trabajar en un sistema, va a pasar que en algún momento vamos a necesitar convertir el tipo de datos con el que estamos trabajando.

En lenguajes de tipado fuerte, como hemos hablado, el tipo de datos es fundamental. En el caso de java el compilador puede hacer la conversión automática si necesitamos hacer la conversión de un tipo de dato de menor tamaño a uno de mayor tamaño.

Si queremos hacer de manera inversa: de mayor a menor tamaño. El compilador nos marcará un error de compilación

Por lo tanto, pongamos la atención en la conversión automática. Por ejemplo si hacemos esto en nuestra IDE:

```
int valor1 = 3;
double valor2 = valor1;

System.out.print("Mostramos el valor convertido: " + valor2);
```

Nos va imprimir por consola: 3.0

Tomó el valor entero 3 y lo convirtió a un número real 3,0.

Lo cual está perfecto

Ahora bien. Hay situaciones donde la conversión no se ajusta a las necesidades reales a la hora de hacer un sistema.

Hagamos esta operación:

```
int numero1 = 19; <br>
int numero2 = 2;

double resultado = numero1 / numero 2;

System.out.print("El resultado de la división es: " + resultado);
```

Ejecutamos el programa y nos va a devolver por consola: 9.0

Ya vimos que el tipo de dato double maneja número con coma, entonces ¿Por qué el resultado es incorrecto, sabemos que es 9,5?

La respuesta a esa pregunta es porque el compilador no puede manejar automáticamente la transformación de tipo int a tipo double (o byte a int, int a float, etc) cuando viene de una operación en base a dos números enteros.

Esto se debe a que los lenguajes crean sus datos gracias a estándares que se consensúan internacionalmente.

¡NO es algo que a alguien se le ocurre y listo!

Recordemos lo que vimos sobre tipos de datos primitivos.

Cuando nosotros decimos que vamos a crear un dato del tipo int, el compilador, basándose en esos estándares construye el dato y lo guarda en memoria. A partir de ese momento el dato ya existe de una forma determinada.

Pensemos en esta analogía:

Imaginemos que tenemos dos ficheros donde guardamos todos los nombres de las personas que conocemos en orden alfabético.

En el primero tenemos los nombres de la **A a la M** y en el segundo tenemos los nombres de la **N a Z**.

Después creamos otro fichero donde vamos a guardar, también organizado alfabéticamente, todos los nombres de la **A a la Z**.

Ahora bien, viene una compañera o compañero y nos pide los primeros 100 nombres. Nosotros se los damos tomándolos, obviamente, del tercer fichero.

Pero cuando nuestra compañera o compañero lo va a guardar en el fichero que le dieron descubre que su fichero **está organizado en grupos**. Por ej: Familiares, Amigos, Trabajo, etc.

Queda claro que si no lo organiza antes de guardarlo va a tener todos los contactos desordenados y sería inservible.

Volvamos a nuestro ejemplo y hagamos esto:

```
int numero1 = 19;
int numero2 = 2;

double resultado = (double) numero1 / número 2;

System.out.print("El resultado de la división es: " + resultado);
```

Si ejecutamos el programa ahora vamos a tener el resultado que necesitamos.

Este proceso de forzar al compilador para que convierta el tipo de dato, antes de guardarlo, se le llama **conversión de tipo de datos** y coloquialmente lo llamamos "castear" o "casteo"

En resumen, castear es la manera que tenemos en Java de decirle al compilador cómo convertir un dato para que encaje en otro tipo y así tratar de evitar la pérdida de datos.

MUY IMPORTANTE: Lo que hay que entender es que siempre tenemos el mismo tamaño de datos. Recordemos lo que vimos en la clase de tipos de datos primitivos. Cuando creábamos un int, estábamos creando un dato de 32 bits. Y cuando creábamos un float también estábamos creando un dato de 32 bits.

→ Solo cambia la forma de organizar esos bits.

5.1.2 Clase Scanner

La Clase Scanner forma parte del paquete java.util. O sea es un elemento dentro de ese paquete que contiene muchas otras utilidades. Algo muy práctico a la hora de programar.

¿Para que nos sirve esta utilidad?

Principalmente se usa para interactuar con el usuario para que este ingrese datos a nuestro programa.

Por ejemplo: Tenemos que calcular el área de un cuadrado. Sabemos que la fórmula es $L \times L$ (lado por lado), pero no nos dan el valor sino que nos dicen que se lo pidamos al usuario.

¿Como hacemos eso?

Tenemos que crear una variable usando la Clase Scanner.

Hacemos en nuestra IDE dentro de nuestra función Main:

```
Scanner ingreso = new Scanner(System.in);<br>
System.out.println("Ingrese el valor del lado del cuadrado ");<br>
int lado = ingreso.nextInt();

int area = lado * lado;

System.out.println("El area del cuadrado es: " + area);
```

Si ejecutamos el programa vamos a ver en consola que el programa queda detenido en el mensaje:

Ingrese el valor del lado del cuadrado

Eso es porque está esperando la intervención del usuario.

Si hacemos clic con el mouse en la consola nos va aparecer un prompt (van a ver esté símbolo | parpadeando)

Ahí ya pueden ingresar un número y al presionar enter el programa va calcular el área, lo va a mostrar por la consola y va a terminar.

Esta Clase permite la entrada por teclado distintos tipos de datos.

(usamos en este caso la misma variable que creamos en primer ejemplo: Scanner ingreso = new Scanner(System.in);)

Algunos ejemplos son:

Para textos:

```
String nombre = ingreso.next()<br>
```

Para un número float:

```
float numFloat = ingreso.nextFloat();<br>
```

Para un número double:

```
double numDouble = ingreso.nextDouble();<br>
```

5.2 Ejercicios para la clase 03

En esta clase vimos variables, constantes y expresiones.

Primeros pasos para estos ejercicios:

1. Abrir la IDE (Netbeans)
 2. Crear un nuevo programa (o abrir uno existente)
 - a. Si estás creando uno nuevo, acordate de seleccionar la opción "Java with Ant".
-

5.2.1 `java.util.Scanner`

5.2.2 Ejercicio 11: Área de un círculo

Pedir que se ingrese el valor del radio `r`, calculá el área con la fórmula: $A = \pi \times r^2$

5.2.3 Ejercicio 12: Perímetro de un cuadrado

Ingresar el valor del lado (`l`), calculá el perímetro con la fórmula: $4 \times l$.

5.2.4 Ejercicio 13: Velocidad promedio

Tenés una distancia (`distanciaKm`) y un tiempo (`horas`). Calculá la velocidad promedio en km/h.

5.2.5 Ejercicio 14: Minutos a horas y minutos

Tenés un número entero de minutos. Convertilo a horas y minutos. Ejemplo: 130 minutos → 2 horas y 10 minutos.

5.2.6 Ejercicio 15: Dígito de las unidades

Ingresar por teclado el valor de un número entero `num`, obtené el dígito de las unidades con `num % 10`.

5.2.7 Ejercicio 16: División entera y resto

Pedir el valor de dos variables enteras `a` y `b`, mostrá el cociente entero y el resto.

5.2.8 Ejercicio 17: Potencia

Pedir el valor de dos variables enteras base y exponente, calculá la potencia

5.2.9 Ejercicio 18: Kilogramos a libras

Convertí un peso en kilogramos (kg) a libras usando la fórmula: $libras = kg \times 2.20462$.

5.2.10 Ejercicio 19: Promedio de notas con decimales

Pedir el ingreso de 4 notas. Calculá el promedio y mostrálo con 2 decimales

5.2.11 Ejercicio 20: Doble y triple de un número

Pedir que se ingrese el valor de un número entero, calculá y mostrá el doble y el triple.

5.2.12 Ejercicio 21: Área y perímetro de un triángulo equilátero

Sabiendo el valor de un lado `a`

1. Calculá el perímetro con la fórmula: $P = a \times 3$
2. Calculá el área. Con la fórmula: $A = a^2 \times \frac{\sqrt{3}}{4}$

5.2.13 Ejercicio 22: Conversión de metros a cm y mm

Pedir que se ingrese el valor de una longitud en metros (double), convertila a centímetros y milímetros.

5.2.14 Ejercicio 23: Promedio de velocidad en metros por segundo

Convertí una velocidad en km/h (`vKmH`) a m/s (`vMs`) $vMs = vKmH \times 1000 \div 3600$.

5.2.15 Ejercicio 24: Descuento en un producto

Pedir que se ingresen los valores de un precio y un porcentaje de descuento, calculá el precio final.

5.2.16 Ejercicio 25: Tiempo de viaje

Pedir que se ingrese el valor de una distancia en km y una velocidad en km/h, calculá el tiempo de viaje en horas.

5.2.17 Ejercicio 26: Concatenación de edad

Dado nombre y edad, mostrá el mensaje "Me llamo y tengo años."

5.2.18 Ejercicio 27: Interés simple

Calculá el interés simple con la fórmula: $I = \text{capital} * \text{tasa} * \text{tiempo}$.

5.2.19 Ejercicio 28: Interés compuesto

Calculá el monto final con la fórmula: $M = \text{capital} * (1 + \text{tasa})^{\text{tiempo}}$.

5.2.20 Ejercicio 29: Conversión de dólares a pesos

Pedir que se ingrese un monto en dólares y un tipo de cambio, calculá el equivalente en pesos.

5.2.21 Ejercicio 30: Conversión de segundos a horas:minutos:segundos

Pedir que se ingrese el valor de un entero segundosTotales, convertilo a formato `hh:mm:ss`.

5.2.22 Ejercicio 31: Promedio ponderado

Tenés tres notas: `n1`, `n2`, `n3` con pesos 0.2, 0.3 y 0.5 respectivamente. Calculá el promedio ponderado.

5.2.23 Ejercicio 32: Promedio de edad de un grupo

Pedir el ingreso de la edad de 4 personas en variables y calculá el promedio.

5.2.24 Ejercicio 33: Separar decenas y unidades

Pedir que se ingrese el valor de un número de dos cifras, obtené la decena y la unidad. Por ejemplo, si el número entero es 23 la decena es el 2 y la unidad es el 3.

6. Clase 04: Arrays

6.1 Clase 04: Arrays (vectores y matrices)

6.1.1 Definición

Los arrays en java son una colección ordenada, indexada y de tamaño fijo. Cuyos elementos pueden cambiar de valor pero no de tipo.

Esto que a primera vista es complejo y confuso lo vamos a ir desgranando paso a paso para entenderlo.

6.1.2 Vectores (Arrays unidimensionales)

Empecemos por este tipo de array para adentrarnos en estos conceptos.

Imaginemos que tenemos un estante donde ubicamos una serie de cajas, por ej: 10 cajas. Es importante que las cajas sean iguales y pueden contener un solo tipo de elemento, por ejemplo: zapatillas. Debajo de cada caja vamos a numerar el orden de las cajas.

Acá hagamos una aclaración: Naturalmente si nosotros les decimos que numeren el orden de las cajas, ustedes harían: **1, 2, 3, ... 10**, pero para nuestro ejemplo lo vamos a hacer iniciando en 0. O sea, nos quedará: **0, 1, 2, 3, ... 9**.

¿Por qué es esto?

Porque tenemos que recordar que el 0 para nosotros es un valor (existe) y siempre lo tomamos como el primer número entero positivo.

Si volvemos a nuestro ejemplo nos quedaría algo parecido a la imagen:

Pasemos a trabajar con un Array unidimensional en java.

Crear un vector no es otra cosa que crear una variable y declararla de una manera que el compilador entienda que lo que estamos creando es un array.

A lo que ya sabemos en la declaración de una variable del tipo entera:

```
int numero = 4;
```

La modificamos de esta manera:

```
int[] numeros = new int[3];
  ^      ^      ^      ^
  TIPO   ID     OP     VALOR
```

Como podemos ver, a el formato que aprendimos al declarar variables le hicimos algunos cambios.

Analicemos lo que cambió:

En el TIPO: Le agregamos el modificador [] (corchete de apertura y de cerrado)

Ese modificador le dice al compilador que estamos creando un array.

Y finalmente después del operador relacional "=" vemos que para el valor usamos: **new int[3]**;

Si aprovechamos la analogía de nuestro estante, estamos diciendo:

Vamos a crear un estante nuevo con 3 cajas.

Si ajustamos nuestro código a la analogía de nuestro estante haríamos esto:

```
int[] numeros = new int[10];
```

Creamos un estante con 10 cajas para guardar enteros.

Es importante entender que solo estamos creando el estante con las cajas y su número identificador, pero todavía no hemos puesto números adentro de esas cajas.

Tendríamos esto:

```
Caja 0 -> vacía
Caja 1 -> vacía
Caja 2 -> vacía
Caja 3 -> vacía
...
Caja 9 -> vacía
```

Ahora bien. Lo que necesitamos ahora es llenar esas cajas.

Pasando a nuestro código lo hacemos así:

```
int[] numeros = new int[4];

numeros[0] = 25;
numeros[1] = 2;
numeros[2] = -8;
numeros[3] = 123;
```

Como podemos ver usamos el identificador del array con el número de orden (índice) entre corchetes para que el compilador sepa exactamente donde ubicar el valor.

O sea en la primera caja, que tiene como índice 0, le ponemos el valor 25 adentro.

En la segunda caja, con el índice 1, le ponemos el valor 2 adentro.

En la tercera caja, con el índice 2, le ponemos el valor -8 adentro.

En la cuarta caja, con el índice 3, le ponemos el valor 123 adentro.

Si queremos ver por consola el contenido de algún elemento del arrays, hacemos

```
System.out.println(numeros[0]); // muestra el contenido de la caja 0 → imprime 25
System.out.println(numeros[1]); // muestra el contenido de la caja 1 → imprime 2
System.out.println(numeros[2]); // muestra el contenido de la caja 2 → imprime -8
System.out.println(numeros[3]); // muestra el contenido de la caja 3 → imprime 123
```

Ahora cambiemos algún valor de los ya asignados a nuestro array

Si hacemos esto

```
numeros[2] = 1000;
```

Cuando lo mostremos por consola, veremos esto:

```
System.out.println(numeros[0]); // muestra el contenido de la caja 0 → imprime 25
System.out.println(numeros[1]); // muestra el contenido de la caja 1 → imprime 2
System.out.println(numeros[2]); // muestra el contenido de la caja 2 → imprime 1000
System.out.println(numeros[3]); // muestra el contenido de la caja 3 → imprime 123
```

Por último veamos otra manera de declarar un array.

Podemos crear el array y darle valores a cada índice en la misma declaración. Lo único que cambia es la forma de darle valor después del operador relacional

```
int[] numeros = {55, 1, 0, 29, 44, 100};
```

En este caso estamos creando un estante con 6 cajas y al mismo tiempo las llenamos con esos valores.

En la primera caja, que tiene como índice 0, le ponemos el valor 55 adentro.

En la segunda caja, con el índice 1, le ponemos el valor 1 adentro.

En la tercera caja, con el índice 2, le ponemos el valor 0 adentro.

En la cuarta caja, con el índice 3, le ponemos el valor 29 adentro.

En la quinta caja, con el índice 4, le ponemos el valor 44 adentro.

En la sexta caja, con el índice 5, le ponemos el valor 100 adentro.

Al hacerlo de esta manera el compilador entiende que cada coma separa un valor de otro.

6.1.3 Matrices (Arrays bidimensionales)

Ahora ampliamos la analogía que usamos para el vector.

Vamos a crear una estantería en lugar de solo un estante. Entonces vamos a tener varios estantes del mismo tamaño

0	Z	Z	Z	Z	Z	Z	Z	Z	Z
	0	1	2	3	4	5	6	7	8
1	Z	Z	Z	Z	Z	Z	Z	Z	Z
	0	1	2	3	4	5	6	7	8
2	Z	Z	Z	Z	Z	Z	Z	Z	Z
	0	1	2	3	4	5	6	7	8
3	Z	Z	Z	Z	Z	Z	Z	Z	Z
	0	1	2	3	4	5	6	7	8

Fíjense que ahora cada estante tiene otro índice.

Para que lo veamos claramente lo pusimos en color verde.

Tenemos 4 filas (los números en verde) y cada fila tiene 10 columnas (los números en celeste)

En nuestra estantería armamos un primer estante (0, recordemos que nuestro primer número natural es 0, no 1) con 10 cajas.

De la misma manera para el segundo: Tendrá el índice 1 y 10 cajas también.

Así hasta el último estante = 3.

Es importante recordar que no puede hacer un estante con 3 cajas. Todos los estantes tienen el mismo tamaño.

Vayamos a como lo declaramos nuestra estantería en java.


```
int[][] matriz = new int[4][10];
```

Fíjense que es similar al vector, solo le agregamos otra dimensión.

En el vector hacíamos, por ejemplo

```
int[] vector = new int[10];
```

Solo decimos cuantos lugares va a tener el estante. En este caso va a tener 10 lugares.

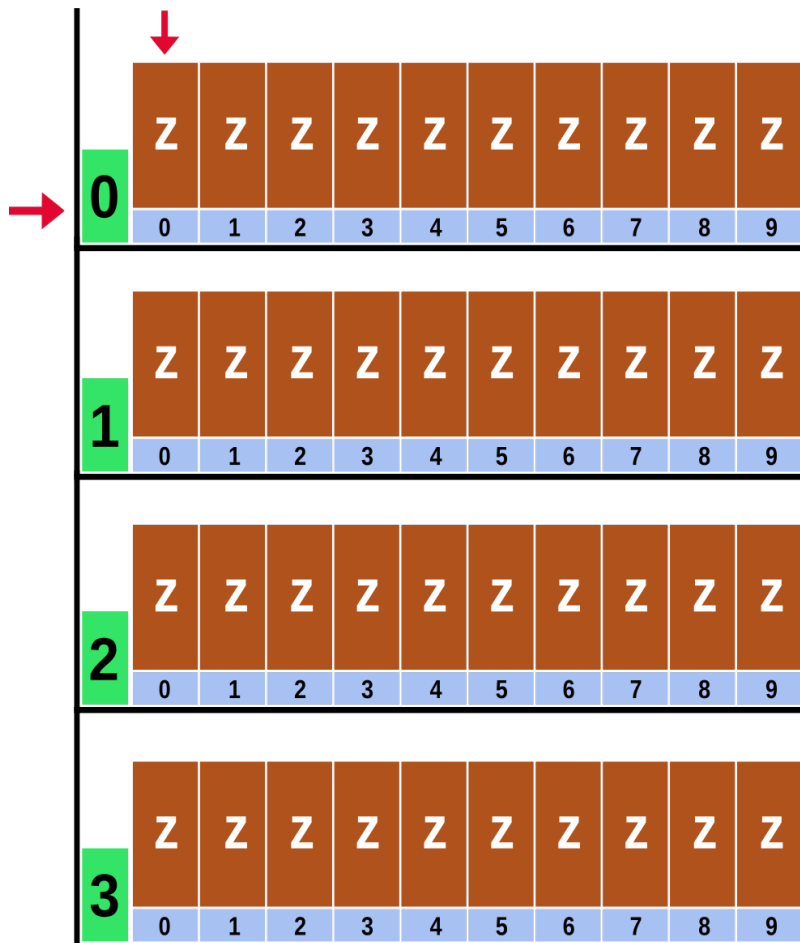
En la matriz le agregamos cuantas filas va a tener.

Esa nueva dimensión va antes de lo que teníamos, por eso [4] va antes de las columnas:

```
int[][] matriz = new int[4][10]
```

Ahora hablemos de como el asignamos valores. Como hay dos dimensiones si queremos darle valor a nuestra matriz tenemos que especificar a qué fila y qué columna nos referimos.

Si queremos cargar un valor en el primer lugar. Nuestra primer caja de nuestro primer estante



Lo hacemos así:

```
matriz[0][0] = 22;
```

Es la fila 0 y la columna 0.

Si queremos llenar más cajas de nuestro primer estante tenemos que usar la referencia al estante y vamos cambiando la caja donde guardar el valor.

Por ej: el primer estante con la segunda y tercer caja.

```
matriz[0][1] = 13;
matriz[0][2] = 8;
```

¿Y si queremos guardar en la décima caja?

```
matriz[0][9] = 2;
```

¿Y si queremos guardar en la cuarta caja del tercer estante?

```
matriz[2][3] = 55;
```

De manera similar si lo queremos mostrar por consola. Tenemos que hacer referencia a los dos índices: el de la fila y el de la columna.

```
System.out.println("Mostramos el último valor que agregamos: " + matriz[2][3]);
```

Ahora vamos a declarar la matriz y asignarle valores a las posiciones, lo hacemos de esta manera.

```
int [ ][ ] numeros = {{72,55,89},{21,3,1}};
```

Separemos por grupos para ver como se construye la asignación de valores.

Tomemos la parte de asignación de valores -> `{{72, 55, 89}, {21, 3, 1}}`

Las primeras llaves `{ }` corresponden al bloque que corresponde a la matriz completa.

Las llaves interiores separadas por la coma (,) agrupan los valores por fila y columna.

Por lo tanto `{72, 55, 89}` son los valores de la primera fila, la que tiene índice 0 y las columnas de esa fila.

```
Fila 0 → Columna 0 = 72
Fila 0 → Columna 1 = 55
Fila 0 → Columna 2 = 89
```

El otro grupo `{21,3,1}` son los valores de la segunda fila, con índice 1 y sus correspondientes columnas.

```
Fila 1 → Columna 0 = 21
Fila 1 → Columna 1 = 3
Fila 1 → Columna 2 = 1
```

La impresión por consola es igual a lo que ya hemos hecho.

6.1.4 Matrices (Arrays tridimensionales)

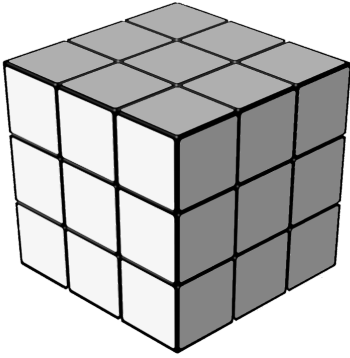
Algo importante a tener en cuenta es que se puede agregar una capa más.

Este escenario es del caso de la programación y diseño 3D. Por ejemplo: Unity.

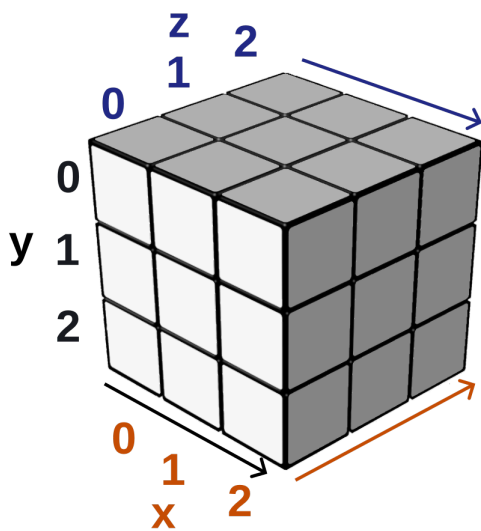
Unity es un motor de videojuegos que se programa y diseña en 3D.

Vayamos a ver un ejemplo.

Supongamos que nuestra estantería es como un cubo: 3 estantes con 3 cajas en cada columna y 3 filas de cajas en cada estante.



Para poder usarlo como figura de referencia tenemos que determinar sobre que lados vamos a identificar nuestras filas, columnas y la profundidad del cubo.



O sea todos los lugares del cubo se van a identificar de está forma general:

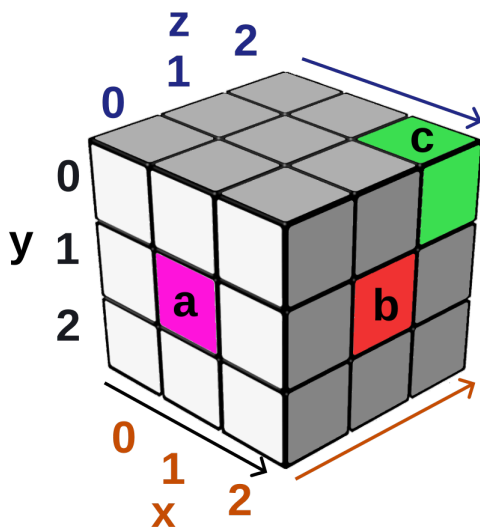
```
cubo[x][y][z]
```

En la figura marcamos cada eje, su índice y la "dirección" en la que tiene su dominio con un mismo color.

El eje X es en color naranja, el eje Y en negro y el eje Z en azul.

Ahora marquemos 3 lugares de nuestro cubo para poder determinar sus coordenadas.

Veamos la siguiente figura:



Podemos ver los lugares donde están a, b y c.

Para el lugar "a" vemos que está en línea con la coordenada 1 de X. La coordenada 1 de Y. Pero no tiene desplazamiento sobre el eje Z, está al frente del cubo. O sea corresponde con la coordenada 0 del eje Z.

Obviando por ahora la sintaxis de java podemos identificar el punto a de la siguiente manera: `cubo[1][1][0]`

Analicemos el punto "b". Sobre el eje X está en la coordenada 2. Sobre el eje Y está en la coordenada 1 y tiene la coordenada 1 del eje Z.

Por lo tanto lo podemos identificar así: `cubo[2][1][1]`

Por último tenemos el punto "c" que su coordenada X es 2, su coordenada Y es 0 y la coordenada Z es 2.

Nos quedaría así: `cubo[2][0][2]`

Ahora pasemos a como declaramos una matriz tridimensional en java.

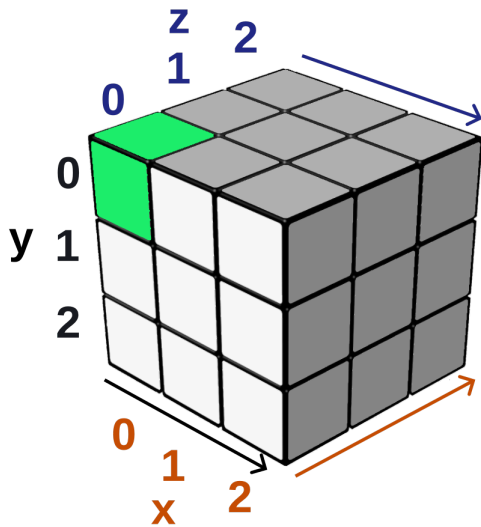
Sigamos con el tamaño de la matriz que tenemos en la figura, la declaramos así

```
int[ ][ ][ ] matriz3D = new int[3][3][3];
```

Recuerden que al igual que en el caso del vector y de la matriz bidimensional en este tipo de declaración hacemos una matriz nueva indicando solo el tamaño. No le estamos asignando valores a la matriz.

Para asignarle valores hacemos parecido a lo que ya hicimos en los casos anteriores solo que necesitamos indicar las 3 coordenadas.

Vamos a ingresar un valor en la primera ubicación de la matriz. La 0, 0, 0.

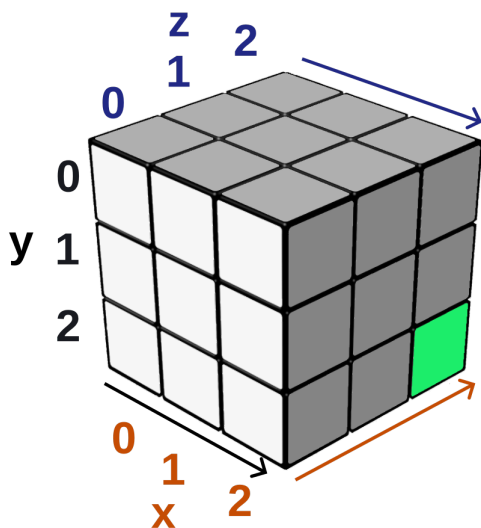


```
matriz3D[0][0][0] = 77;
```

Si por el contrario queremos ingresar un valor en la última ubicación de la matriz, sería:

```
matriz3D[2][2][2] = 134;
```

Y en la figura sería el punto que está al fondo y abajo a la derecha.



Ahora nos falta crear la matriz y asignar valores al mismo tiempo.

Como referencia tenemos que pensar que necesitamos un par de llaves de apertura y de cierre por cada capa que agreguemos a nuestra matriz. Nuestra matriz, la de la figura es de $3 \times 3 \times 3$. Al crearla vamos a tener esto:

```
int[][][] cubo = { // Está llave representa al contenedor más grande que sería el cubo en sí

    { // Está llave es la apertura de todas las posiciones de la fila 0

        {72,55,89}, {21,3,1}, {18,19,4}

    },
    { // Está llave es la apertura de las posiciones correspondiente a la fila 1

        {33,5,6}, {12,0,8}, {129,65,113}

    },
    { // Está llave es la apertura de las posiciones correspondiente a la fila 2

        {77,54,7}, {88,11,71}, {54,7,10}

    }

};
```

Tomemos este trozo de código:

```
{72,55,89}, {21,3,1}, {18,3,4}
```

Aquí tenemos que **cada coma entre las llaves separa las columnas y cada número separado por coma es la posición correspondiente a la profundidad: El eje Z**

Por ejemplo si quiero imprimir por consola el valor 129 que está en esta parte del código:

```
{ // Está llave es la apertura de las posiciones correspondiente a la fila 1

    {33,5,6}, {12,0,8}, {129,65,113}

    // Columna0, Columna1, Columna2
},
```

Lo haríamos así:

```
System.out.println("Valor que se encuentra en el lugar [1][2][0]: " + cubo[1][2][0]);
```

6.1.5 En resumen

Definición técnica en Java

Array (vector 1D):

1. Ordenado en una dimensión
2. Tamaño fijo (cantidad de elementos)
3. Acceso indexado con 1 índice
4. Mutable en contenido

Array bidimensional (matriz 2D):

1. Ordenado en dos dimensiones (filas y columnas)
2. Tamaño fijo (cantidad de filas y de columnas)
3. Acceso indexado con 2 índices [fila][columna]
4. Mutable en contenido

Array tridimensional (cubo 3D):

1. Ordenado en tres dimensiones (ejes X, Y y Z)
2. Tamaño fijo en cada dimensión
3. Acceso indexado con 3 índices $[x][y][z]$
4. Mutable en contenido
5. Puede pensarse como un conjunto de matrices 2D apiladas

6.2 Ejercicios para la clase 04

En esta clase vimos variables, constantes y expresiones.

Primeros pasos para estos ejercicios:

1. Abrir la IDE (Netbeans)
2. Crear un nuevo programa (o abrir uno existente)
 - a. Si estás creando uno nuevo, acordate de seleccionar la opción "Java with Ant".

6.2.1 Ejercicio 01

Diseñar una aplicación que solicite al usuario que ingrese un número, y mostrar si ese número es par o impar.

7. Clase 05: Estructuras de control

7.1 Estructuras de control de flujo

Cuando programamos, nuestro programa sigue un camino predeterminado: a eso lo llamamos **control de flujo**.

En condiciones normales, este flujo es **secuencial**, es decir, el programa empieza en la primera instrucción y continúa en el mismo orden en el que escribimos las sentencias en la IDE. Veamos un ejemplo:

```
String miNombre = "Juan";
String miApellido = "Fernandez";

System.out.println("Hola ¿cómo estás? " + miNombre + " " + miApellido);
```

Y ejecutamos vemos por consola:

```
Hola ¿como estas? Juan Fernandez
```

Si cambiamos el orden:

```
String miApellido = "Fernandez";
String miNombre = "Juan";

System.out.println("Hola ¿cómo estás? " + miNombre + " " + miApellido);
```

La salida por consola será:

```
Hola ¿como estas? Fernandez Juan
```

👉 Como se ve, el flujo del programa fue **sentencia a sentencia, una después de la otra**, sin que nosotros lo controlemos de manera especial.

En clases anteriores usamos la clase **Scanner** para leer datos del usuario. Por ejemplo:

```
Scanner ingreso = new Scanner(System.in);

System.out.println("Ingrese su nombre: ");
String miNombre = ingreso.next();
System.out.println("Hola ¿cómo estás? " + miNombre);
```

Cuando lo ejecutamos, la consola muestra:

Ingrese su nombre:

El programa se detiene y espera que el usuario escriba un valor y presione Enter.

Cuando lo hacíamos y apretábamos enter

El flujo continúa:

Ingrese su nombre:

Juan

Hola ¿cómo estás? Juan

Esto nos enseña que un programa no siempre avanza de manera automática: a veces depende de datos externos o de condiciones.

Por lo tanto, ya podemos deducir que un programa no siempre sigue un único camino. A veces necesita **tomar decisiones** o **esperar una intervención externa**.

Las **estructuras de control** son justamente las que permiten al programa **elegir qué camino tomar**.

7.1.1 Estructuras condicionales

Tomemos un ejemplo de la vida real:

Si está lloviendo, salgo con paraguas; si no, salgo sin paraguas.

En programación, esto se traduce en una estructura if – else.

Lo podemos representar en pseudocódigo o de manera casi coloquial

```
Si (llueve) entonces
    salgo con paraguas
Sino
    salgo sin paraguas.
```

Si analizamos lo que escribimos arriba tenemos

1. → “Si” Es la declaración inicial de la condición.
2. → (llueve) Esto es la condición que va a evaluar.
3. → “entonces” Sería qué hacer si está lloviendo.
4. → “salgo con paraguas” La acción que hará el programa si efectivamente está lloviendo.
5. → “Sino” Declaración si la evaluación no se cumple.
6. → “salgo sin paraguas.” Acción que hará el programa si no está lloviendo.

Es importante entender que la sentencia inicial

→ ***Si (llueve) entonces***

evalúa si la condición que está entre paréntesis se cumple, para nosotros: si es verdadera.

Hagamos un ejemplo en nuestro código

```
boolean llueve = true;

if (llueve) {
    System.out.println("Salgo con paraguas");
} else {
    System.out.println("Salgo sin paraguas");
}
```

Cuando lo ejecutamos nos va a mostrar por consola:

→ ***Salgo con paraguas***

Si cambiamos el valor de llueve a false

```
boolean llueve = false;

if (llueve) {
    System.out.println("Salgo con paraguas");
} else {
    System.out.println("Salgo sin paraguas");
}
```

Al ejecutar el programa vemos por consola:

→ ***Salgo sin paraguas***

👉 Acá conviene entender bien el concepto de que evalúa que se cumpla la condición.

¿Qué significa “se cumple la condición”?

Cuando hablamos de que una condición se cumple, queremos decir que la **expresión dentro de los paréntesis del if se evalúa como true**.

Si es true, se ejecuta el bloque del if.

Si es false, se ejecuta el bloque del else (si lo hubiera).

7.1.2 Operador de negación !

Podemos usar el operador ! (**NOT**) para negar una condición.

Ilustremos con un ejemplo:

```
boolean estado = true;

if (!estado) {
    System.out.println("Salgo con paraguas");
} else {
    System.out.println("Salgo sin paraguas");
}
```

En este caso, estado es true, pero !estado significa "NO verdadero", es decir, false.

Por lo tanto, el programa muestra:

Salgo sin paraguas

Ahora cambiemos estado al valor = false

```
boolean estado = false;

if (!estado) {
    System.out.println("Salgo con paraguas");
} else {
    System.out.println("Salgo sin paraguas");
}
```

En este ejemplo **estado** ya viene con valor falso, por lo tanto como nosotros estamos evaluando que NO sea verdadero, el programa va evaluar que se cumple la condición y va a mostrar el primer mensaje

→ **Salgo con paraguas.**

Tenemos que acostumbrarnos que cuando se habla de que se cumple la condición nos referimos a que es verdadera. O sea que la condición que evaluamos es verdadera, más allá del valor que tiene desde antes los elementos que se evalúan.

Para recordar

1. Una condición es una expresión que se evalúa como true o false.
2. El if ejecuta el bloque de código solo si la condición es verdadera.
3. El else se ejecuta cuando la condición es falsa.
4. El operador ! invierte el valor de la condición.

7.2 Ejercicios para la clase 05

En esta clase vimos variables, constantes y expresiones.

Primeros pasos para estos ejercicios:

1. Abrir la IDE (Netbeans)
2. Crear un nuevo programa (o abrir uno existente)
 - a. Si estás creando uno nuevo, acordate de seleccionar la opción "Java with Ant".

7.2.1 Ejercicio 01

8. Clase 06: Funciones II

8.1 Clase 06: Funciones segunda parte

8.2 Ejercicios para la clase 06

En esta clase vimos variables, constantes y expresiones.

Primeros pasos para estos ejercicios:

1. Abrir la IDE (Netbeans)
2. Crear un nuevo programa (o abrir uno existente)
 - a. Si estás creando uno nuevo, acordate de seleccionar la opción "Java with Ant".

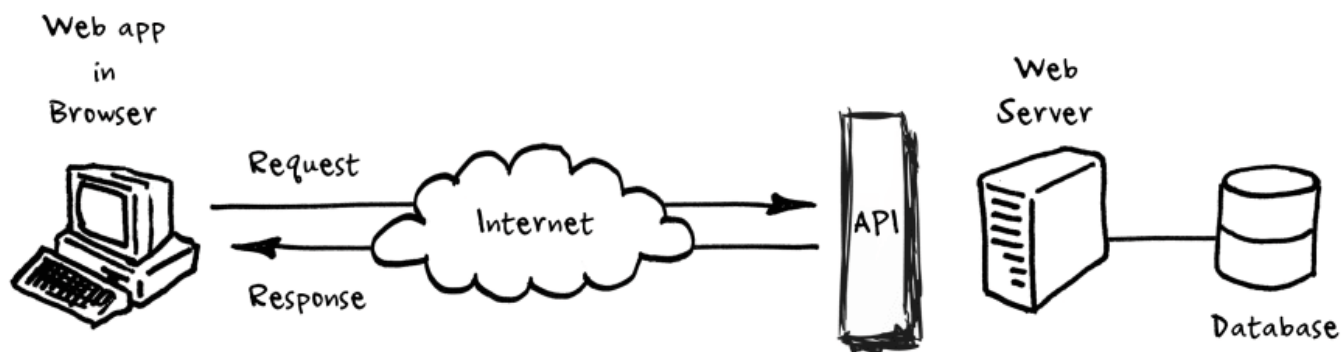
8.2.1 Ejercicio 01

9. Clase 07: API de Java

9.1 Clase 07: API de Java

El término API viene del inglés *Application Programming Interface* y lo podríamos traducir como Interfaz de Programación de Aplicaciones.

Ejemplo del mozo como la API entre servidor (cocina) y cliente. Pedido del cliente (request), platos (response).



9.2 Ejercicios para la clase 07

En esta clase vimos variables, constantes y expresiones.

Primeros pasos para estos ejercicios:

1. Abrir la IDE (Netbeans)
2. Crear un nuevo programa (o abrir uno existente)
 - a. Si estás creando uno nuevo, acordate de seleccionar la opción "Java with Ant".

9.2.1 Ejercicio 01

10. Clase 08: Clases

10.1 Clase 08: Clases

10.2 Ejercicios para la clase 08

En esta clase vimos variables, constantes y expresiones.

Primeros pasos para estos ejercicios:

1. Abrir la IDE (Netbeans)
2. Crear un nuevo programa (o abrir uno existente)
 - a. Si estás creando uno nuevo, acordate de seleccionar la opción "Java with Ant".

10.2.1 Ejercicio 01

11. Clase 09: Arrays

11.1 Clase 09: Arrays (o vectores)

11.2 Ejercicios para la clase 09

En esta clase vimos variables, constantes y expresiones.

Primeros pasos para estos ejercicios:

1. Abrir la IDE (Netbeans)
2. Crear un nuevo programa (o abrir uno existente)
 - a. Si estás creando uno nuevo, acordate de seleccionar la opción "Java with Ant".

11.2.1 Ejercicio 01

12. Glosario

13. Ejercicios

13.1 Ejercicios sueltos



<https://github.com/anairamzap/curso-polito>