PROGRAMACIÓN ESTRUCTURADA Y MODULAR. ENTORNO DE DESARROLLO. PROGRAMAR EN JAVA.

1. ENTORNO DE DESARROLLO.

Antes de entrar en JAVA vamos a ver que es un entorno de desarrollo, cual vamos a utilizar, cómo instalarlo y las funciones básicas.
Como hemos visto en la unidad didáctica 1 un entorno de desarrollo es una herramienta de desarrollo de software utilizado para escribir, generar, probar y depurar un programa.
En mi caso voy a usar NetBeans. Os dejo un enlace sobre tutorial de cómo instalarlo y configurarlo

2. CARACTERÍSTICAS DE JAVA.

Java es un lenguaje de programación creado en 1995. Este lenguaje cuenta con las siguientes características:

- Orientado a Objetos (OO). Más adelante profundizaremos en qué son los objetos.
- Independencia de la plataforma => hardware o sistema operativo.
    - Basado en estándares (Java community Process)
    - Fácil de aprender y usar (toma lo mejor de otros lenguajes orientados a objetos, como C++) => usado a nivel mundial.

3. COMENTARIOS.

Cuando estamos programando existe la opción de escribir comentarios que nos van a permitir aclarar qué estamos haciendo. En java hay distintas formas de poner comentarios:

```
//Comentario de una sola línea
/*Comentario de
más de una línea*/

/**Comentario de documentación*/   => generación automática.
```

4. DATOS.

En Java hay dos tipos de datos:

- Primitivos, heredados de lenguajes no O.O. y que no tiene atributos ni métodos;

- Objetos: Antes de entrar en profundidad en ambos, hay que aclarar que cuando identificamos un dato, es decir, le damos nombre, hay ciertas reglas que debemos seguir:
    - Empiezan en letra, $ o guión bajo (_)
    - Combinación de letras y números.
    - Mayúsculas y minúsculas son diferentes.
    - Se puede usar acentos y ñ pero NO es aconsejable.
    - Las clases e interfaces empiezan con mayúscula.
    - Las constantes TODO en mayúsculas.

En Java cuando ponemos el nombre a una variable o constante hay palabras reservadas que no se pueden usar, ya que tienen un uso ya definido. Estas palabras son las siguientes:

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

Tabla 1.

4.1 DATOS PRIMITIVOS, TIPOS.

En la UD1 vimos los tipos de datos de forma básica, pero en java 8 los enteros y los decimales pueden definirse de varias formas dependiendo del tamaño de los mismos. Ver tabla 2.

Tipo	Representación / Valor	Tamaño (en bits)	Valor mínimo	Valor máximo	Valor por defecto
`boolean`	`true` o `false`	1	N.A.	N.A.	`false`
`char`	Carácter Unicode	16	\u0000	\uFFFF	\u0000
`byte`	Entero con signo	8	-128	128	0
`short`	Entero con signo	16	-32768	32767	0
`int`	Entero con signo	32	-2147483648	2147483647	0
`long`	Entero con signo	64	-9223372036854775808	9223372036854775807	0
`float`	Coma flotante de precisión simple Norma IEEE 754	32	±3.40282347E+38	±1.40239846E-45	0.0
`double`	Coma flotante de precisión doble Norma IEEE 754	64	±1.79769313486231570E+308	±4.94065645841246544E-324	0.0

Tabla 2.

Las cadenas o String no son un dato primitivo como tal, son objetos, ver punto 4.4.

4.2 LITERALES.

En Java, un literal es un valor constante que se escribe directamente en el código fuente. Representa un dato fijo que no cambia y que el compilador interpreta como un valor específico de algún tipo de dato.

Los **enteros** se usan para representar números sin decimales y como literales solo tenemos los int y long.

```
int numero = 100;

long grande = 10000000000L; //necesita la letra L (mayúscula preferiblemente para evitar confusión con el número 1), que indica que el número es del tipo long.
```

Si queremos asignar un valor a un byte o un short tendremos que ajustar el rango pero no es necesario, ni permitido, poner una letra al final.

Para los **decimales**, Java tiene:

```
float decimalCorto = 3.14f //usa la letra f o F para indicar que es un float.

double decimalLargo = 2.7182818284 //opcionalmente puede usar la letra d o D, aunque no es obligatoria ya que los literales decimales sin sufijo se interpretan como double por defecto: double ejemplo = 1.23D; es válido.
```

Pero los literales también nos permiten representar distintos sistemas numéricos. Cada sistema tiene un prefijo específico que le indica al compilador

de qué base se trata. Pero ¡ojo! importante, **Java solo permite el sistema decimal (base 10)** para literales de punto flotante. Así tenemos:

- **Decimal (base 10)**
    - Prefijo: ninguno.
    - Es el sistema de numeración que usamos normalmente, con dígitos del 0 al 9.
    - Es el valor por defecto si no se especifica otro prefijo.

```
int numero = 1750;
```

- **Binario (base)**
    - Prefijo: 0b o 0B
    - Utiliza solo los dígitos 0 y 1
    - Introducido a partir de Java 7

```
int binario = 0b1010; // Equivale a 10 en decimal
```

- **Octal (base 8)**
    - Prefijo: 0 (cero)
    - Utiliza los dígitos del 0 al 7
    - ¡Cuidado! Un número que comienza con 0 se interpreta como octal, no decimal.

```
int octal = 012; // Equivale a 10 en decimal
```

- **Hexadecimal (base 16)**
    - Prefijo: 0x o 0X
    - Utiliza los dígitos del 0 al 9 y las letras de la A a la F (mayúsculas o minúsculas), donde: A = 10, B = 11, ..., F = 15

```
int hexadecimal = 0x1A; // Equivale a 26 en decimal
```

En Java, para representar números en notación científica, se emplea la letra e o E seguida del exponente en base 10. Así, `3.5e2` representa el número 350.0 (3.5×10^2) y `2e-3` equivale a 0.002.

Además, los literales numéricos permiten mejorar su legibilidad utilizando el carácter de subrayado (_) entre las cifras, sin que esto afecte a su valor. Por ejemplo: `1_000_000` es equivalente a 1000000, pero resulta más fácil de leer.

¡Ojo! Este caracter **no se puede poner:**

- Al principio o al final del número.
  - `_1000`
  - `1000_`

- Justo antes o después del punto decimal.
  - `3_.1415`
  - `3._1415`

- Inmediatamente antes de un sufijo de tipo (L, F, D).
  - `1000_L`

- Justo antes o después de e o E en notación científica.
  - `1.0_e10`
  - `1.0e_10`

Un **literal de tipo char** representa un único carácter Unicode y se escribe entre comillas simples (' '). Veamos que formas tenemos de representar un literal char en Java:

- **Carácter literal directo:**

```
char a = 'A';
char signo = '?';
char espacio = ' ';
```

- **Caracteres especiales (escape):**

Secuencia de escape	Descripción
\n	Ir al principio de la línea siguiente
\b	Retroceso
\t	Tabulador horizontal
\r	Ir al principio de la línea
\"	Comillas
\'	Comilla simple
\\	Barra inversa

Tabla 3

Ejemplo:

```
char salto = '\n';
char tab = '\t';
```

- **ASCII (valor numérico):**

```
char letra = 65;
System.out.println(letra); // Imprime: A
```

- **Unicode hexadecimal (\uXXXX):** Permite representar cualquier carácter por su código Unicode hexadecimal. Siempre se escriben con 4 dígitos hexadecimales.

```
char letra = '\u0041'; // 'A'
char yen = '\u00A5';   // '¥'
char emoji = '\u263A'; // '☺' (carácter BMP)
```

4.3 CONSTANTES Y VARIABLES.

Ya hemos visto qué son las variables y las constantes. Para definirlas en Java hay que hacerlo de la siguiente forma:

- **Variables.**

  [privacidad] tipo_dato nombre_variable;

  El tipo de dato se asignará a la hora de definir la variable siguiendo la tabla 2. El nombre debe seguir las reglas vistas anteriormente y la privacidad se verá más adelante a qué hace referencia (inicialmente no vamos a poner nada).
  Ejemplos de variables serían…

```
int numero = 2;
float decimal = 2.4f;  /*decimal va con punto bajo. para que reconozca que es flotante poner f, de lo contrario lo considera double.*/

char caracter = 'h';  /* Los caracteres se definen con comillas simples .*/

boolean flag = true;
```

- **Constantes.**

  [privacidad] final tipo_dato NOMBRE_CONSTANTE = valor_constante;

  Se inicializan SIEMPRE cuando se crean. Todo en mayúsculas. Se pone delante **final.**

```
final double PI= 3.1415;
```

4.4 CADENAS O STRINGS.

Las cadenas (String) son un tipo de dato objeto que veremos con más profundidad en la UD3, por ahora debemos saber que los String son un conjunto de caracteres y se definen entre comillas dobles.

```
char variableChar = 'a';
String variableString = "cadena";
```

Una cadena puede contener números pero estos serán tratados como caracteres no como números. Por ejemplo:

```
String num1 = "100";
String num2  = "60";
String num3 = num1+num2;    //esto da "10060" no 160
```

Además, en las cadenas podemos utilizar una serie de secuencias de escape, las cuales empiezan por una barra invertida y siguen con un modificador. *Ver tabla 3 del punto 4.2*.

4.5 CONVERSIÓN DE DATOS.

En Java hay dos maneras de convertir el tipo de un dato en otro:

- **Implícita**: cuando el dato que se desea convertir a un tipo de dato compatible, **simplemente se igualan**. Para poder realizar esta conversión debemos convertir el dato a un tipo mayor que no suponga pérdida de información.

Tipo inicial	Tipo final
byte	short, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

Tabla 4

Por ejemplo:

```
int entero = 12340;
float conversion = entero;
//conversion=12340.0000
```

- **Explícita:** Si la conversión se usa entre tipos no compatibles y representa una pérdida de información, debe realizarse mediante **casting.** Este método sigue la siguiente sintaxis:

variable = (tipo_de_dato) variable_tipo_mayor;

Tipo inicial	Tipo final
short	byte, char
char	byte, short
int	byte, short, char
long	byte, short, char, int
float	byte, short, char, int, long
double	byte, short, char, int, long, float

Tabla 5

Por ejemplo:

```
float n = 100.04f;
int k = (int) n;
char caracter = (char) k; /* caracter =d
```

Hasta ahora hemos hablado de tipos de datos, pero Java es un lenguaje orientado a objetos como se verá más adelante. Por lo que puede ser necesario, en algún momento, usar tanto un tipo de dato primitivo como una clase. Para ello se convertirá mediante una **clase envolvente**:

Tipo primitivo	Clase Envolvente
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Tabla 6

Estas clases tienen métodos (funciones) que pueden ser usadas con ellas pero no con el dato primitivo. Alguno de estos métodos son: parseTipo() o valueOf()
Ejemplos:

```
String cadena2 = "1.23";
float flotante = Float.parseFloat(cadena2);
```

// flotante = 1.23

```
int numero = 728;
String cadena = String.valueOf(numero);
```

cadena = “728”

NOTA: parseTipo lo usamos para convertir una cadena en tipos primitivos y valueOf a clases. Por ejemplo:

```
Integer n1 = Integer.valueOf("623");
int n2 = Integer.parseInt("8237");
```

A modo de resumen tenemos la tabla 7:

CONVERSIONES		
DATO ORIGEN	**DATO SALIDA**	**MÉTODO**
Primitivo	Primitivo	Casting
Objeto	Primitivo	parseTipo()
Primitivo	Objeto	valueOf()
Objeto		

Tabla 7

5. OPERACIONES BÁSICAS.

a. OPERADORES ARITMÉTICOS. TABLA 8.

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
-	operador unario de cambio de signo	`-4`	`-4`
+	Suma	`2.5 + 7.1`	`9.6`
-	Resta	`235.6 - 103.5`	`132.1`
*	Producto	`1.2 * 1.1`	`1.32`
/	División (tanto entera como real)	`0.050 / 0.2` `7 / 2`	`0.25` `3`
%	Resto de la división entera	`20 % 7`	`6`

++	Incremento
	`i++` primero se utiliza la variable y luego se incrementa su valor
	`++i` primero se incrementa el valor de la variable y luego se utiliza
		`4++`
		`a=5;`
		`b=a++;`
		`a=5;`
		`b=++a;`
			`5`
			`a vale 6 y b vale 5`
			`a vale 6 y b vale 6`

--	decremento	`4--`	`3`

Tabla 8.

2. OPERADORES RELACIONALES. TABLA 9.

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
==	igual que	`7 == 38`	`false`
!=	distinto que	`'a' != 'k'`	`true`
<	menor que	`'G' < 'B'`	`false`
>	mayor que	`'b' > 'a'`	`true`
<=	menor o igual que	`7.5 <= 7.38`	`false`
>=	mayor o igual que	`38 >= 7`	`true`

Tabla 9.

3. OPERADORES LÓGICOS. TABLA 10.

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
!	Negación - NOT (unario)	`!false` `!(5==5)`	`true` `false`
|	Suma lógica – OR (binario)	`true | false` `(5==5)|(5<4)`	`true` `true`
^	Suma lógica exclusiva – XOR (binario)	`true ^ false` `(5==5)|(5<4)`	`true` `true`
&	Producto lógico – AND (binario)	`true & false` `(5==5)&(5<4)`	`false` `false`
||	Suma lógica con cortocircuito: si el primer operando es `true` entonces el segundo se salta y el resultado es `true`	`true || false` `(5==5)||(5<4)`	`true` `true`
&&	Producto lógico con cortocircuito: si el primer operando es `false` entonces el segundo se salta y el resultado es `false`	`false && true` `(5==5)&&(5<4)`	`false` `false`

Tabla 10.

Para *&&* es conveniente situar la condición más propensa a ser falsa en el término de la izquierda. Para el operador || es conveniente colocar la condición más propensa a ser verdadera en el término de la izquierda.

Para mejorar el rendimiento de ejecución del código es recomendable emplear en las expresiones booleanas el operador && en lugar del operador & y el operador || en lugar del | para reducir el tiempo de ejecución del programa.

Las operaciones se realizan según el siguiente orden de prioridad:

1. Post-unitarios operadores (expr++, expr--)
2. Pre-unitarios operadores (++expr, --expr)
3. Multiplicación, división, y módulo (, *, /, %)
4. Suma y resta (+, -)

5. Relacionales (<, >, <=, >=)
6. Igualdad (, ==, !=)
7. Operadores lógicos lógicos AND (&&)
8. Operadores lógicos lógicos OR (||)
9. Asignación (=, +=, -=)

6. SENTENCIAS ALTERNATIVAS.

En las sentencias alternativas es necesario las llaves { } cuando la sentencia a cumplir es más de una. En el caso de ser una no es necesario, pero se aconseja ponerlas siempre para que el texto tenga concordancia y prevenir errores futuros en el caso de querer añadir más sentencias.

6.1 SIMPLE.

```
if(condicion){
sentencias;
}
```

Ejemplo:

```
int i;
int j;


/*Si i es menor que j entonces saca por pantalla que i
menor que j, sino no hace nada*/
if (i<j){
     System.out.printf( i + " es menor de" +  j );
 }
```

6.2 DOBLE.

```
if(condicion){
sentencias;
}
else{
sentencias;
}
```

Ejemplo, estructura para ver si un número es par:

```
int numero;
```

```
/*Si el                          módulo(resto) de dividir
el número entre 2 es 0 entonces el número es par*/

if ( numero % 2 == 0 ) {
     System.out.printf( "ES PAR" );
 }

/*Sino es impar*/

 else
 {
            System.out.printf( "ES IMPAR" );
 }
```

3. MÚLTIPLE.

- En el caso de que lo que tengamos sea una EXPRESIÓN, es decir, un tipo de dato, como **byte, short, int, char,** usaremos un **switch.**

```
switch(expresión)
{
// los valores deben ser del mismo tipo de la
expresión
   case valor1 :
      // Sentencias 1;
      break;

   case valor2 :
      // Sentencias 2;
      break;

/*Podemos tener cualquier número de declaraciones de
casos o case*/
/* Por último, podemos poner un caso por defecto que
será lo que se haga si no se cumple ninguno de los
anteriores. */


   default :
      // Sentencias 3;
      break;
}
```

Ejemplo, vemos el valor de un entero y en función de él sacamos por pantalla el día de la semana:

```
int dia;
switch (dia) {
    case 1:
        System.out.println("Lunes");
        break;
    case 2:
        System.out.println("Martes");
        break;
    case 3:
        System.out.println("Miercoles");
        break;
    case 4:
        System.out.println("Jueves");
        break;
    case 5:
        System.out.println("Viernes");
        break;
    case 6:
        System.out.println("Sabado");
        break;
     case 7:
        System.out.println("Domingo");
        break;
 default:
        System.out.println("La semana solo tiene
7dias");
        break;
}
```

Características:

- Usa ":" después de cada *case*.
- Se necesita *break* para evitar que el flujo "caiga" (fall-through) al siguiente caso.
- Permite agrupar varios casos consecutivos.
- No devuelve directamente valores.
- Switch moderno con ->. Desde Java 14 hay una nueva estructura mejorada, para hacer el switch. En este caso usamos "->" en lugar de ":". No hace falta el *break* porque no hay *fall-through.* Además permite varios valores en un mismo caso separados por "," y puede devolver un valor directamente. Veamos algunos ejemplos.

```
int dia = 3;

switch (dia) {
    case 1 -> System.out.println("Lunes");
    case 2 -> System.out.println("Martes");
    case 3 -> System.out.println("Miércoles");
    default -> System.out.println("Fin de semana");
}
```

Si quisiéramos agrupar casos:

```
switch (dia) {
    case 1, 2, 3, 4, 5 -> System.out.println("Día
laboral");
    case 6, 7 -> System.out.println("Fin de semana");
    default -> System.out.println("Día no válido");
}
```

Como expresión (devuelve un valor):

```
String nombreDia = switch (dia) {
    case 1 -> "Lunes";
    case 2 -> "Martes";
    case 3 -> "Miércoles";
    default -> "Desconocido";
};
```

El switch devuelve un resultado que se guarda en una variable. Cuando queramos que un caso devuelva un valor en un case con más de una sentencia se usa la palabra clave "*yield*".

```
String resultado = switch (numero) {
    case 1 -> "Uno";
    case 2 -> {      // Caso con varias instrucciones
        System.out.println("Procesando número 2...");
        yield "Dos";   // yield devuelve el valor al
switch
    }
    default -> "Otro";
};
```

En otras palabras: yield "entrega" un valor desde un bloque de switch.

- En el caso de que estemos valorando una CONDICIÓN usaremos un if-else anidado:

```
if (condición) {
 sentencias;
}
else if (condición) {
    sentencias;
}
else;
```

7. SENTENCIAS BUCLES.

En los bucles es necesario las llaves { } cuando la sentencia a cumplir es más de una, en el caso de ser una no es necesario pero se aconseja poner siempre para que el texto tenga concordancia y prevenir errores futuros en el caso de querer añadir más sentencias.

7.1 While.

Se evalúa la condición y si es **cierta** se realizan las sentencias de dentro. Al ser un bucle estás sentencias se irán repitiendo cada vez que se evalúe la condición y esta sea cierta, por lo que **es necesario que las sentencias de dentro modifiquen la condición o crearemos un bucle infinito**.

```
while (condición) {
    sentencias;
}
```

Ejemplo, valora la variable *suma* (usada anteriormente en el programa) y si esta es menor de 100 pide un número al usuario y sumaselo. Repetir mientras *suma* no alcance el 100.

```
int num;


// mientras suma sea menor de 100 repito
while (suma <100)
{
     /*Pido un nuevo número y lo sumo para modificar
la condición*/
     System.out.println("Por favor introduzca un
número);
```

```
num=sc.nextInt();
            suma=suma+num;
      }
```

7.2 Do… While.

En este caso se hacen las sentencias primero y luego se evalúa la condición, por lo que las sentencias se ejecutarán al menos una vez. Al terminar estas, se evalúa la condición y si es cierta se repiten las sentencias hasta que, en una de las evaluaciones, no sea cierta. Este bucle se suele usar cuando en la primera iteración no conocemos aún el valor de la condición.

Al igual que el MIENTRAS necesitaremos que las sentencias de dentro del bucle modifiquen la condición o nos encontraremos en un bucle infinito.

```
do {
     sentencias;
} while (condición);
```

Ejemplo, pide un número *num, ell valor de este debe ser menor de 10, sino vuelve a pedirlo*.

```
int num;
Scanner sc= new Scanner(System.in);


do {
      /*pedimos un número */
      System.out.println("Introduce un número
menor de 10");
      num=sc.nexInt();


} while (x < 10);
```

7.3 for.

El bucle para se usa cuando conocemos exactamente cuantas veces vamos a realizar el bucle antes de empezarlo. En este caso, en la misma definición del bucle se da un valor inicial para una variable contador, la condición que debe ser verdadera para que se realicen las operaciones y el cambio que sufre la variable en cada iteración.

La variable se puede definir dentro del mismo bucle, al darle el valor inicial. Cuando se hace esto esta variable sólo está definida dentro del bucle. Si se usa fuera no se reconoce.

```
for (inicio; condición; iteración){
    sentencias;
}
```

Ejemplo 1:

```
for (i = valor_inicial; i <= valor_final; i++) {
    sentencias;
}
```

Ejemplo 2: Saca por pantalla una cuenta del 1 al 10.

```
/* bucle for comienza cuando i=1 y corre para i <=10,
x incrementa de 1 en 1 en cada iteración */


for (int i = 1; i <= 10; i++){
     System.out.println(i);
}
```

8. CLASES Y OBJETOS .

8.1 CLASES.

Las clases representan los prototipos de los objetos que tenemos en el mundo real. Es decir, es una generalización de un conjunto de objetos. En la clase es dónde definimos las propiedades y métodos que podrán contener cada una de los objetos.
Por ejemplo, podemos definir una clase que represente las figuras geométricas de los triángulos. Para saber cómo es el triángulo necesitaremos saber su base y altura, además definiremos una serie de funciones u operaciones que se pueden realizar con ello, en este caso calcular el área.

```
class Triangulo {
    private float base;
    private String nombre;
    private float altura;


    public Triangulo(float base, float altura) {
        this.base = base;
        this.altura = altura;
```

```
    }

    public float area() {
        return (base*altura)/2;
    }
}
```

8.2 OBJETOS.

En el punto anterior hemos visto que una clase es una representación genérica de algo que queremos definir. Un objeto parte de una clase y da valores a sus propiedades de manera que ya no son genéricos sino objetos concretos.
Un objeto tiene su estado (o estados) y su comportamiento. Esto se modela mediante propiedades (o variables) y métodos.
Las interacciones con los objetos se hacen mediante **métodos**. Es decir, si queremos conocer información del estado del objeto deberemos de llamar a uno de sus métodos y no directamente a las propiedades. Para llamar a un método habrá que poner:

```
nombre_objeto.metodo();
```

Ejemplos:
Imaginemos una figura geométrica del triángulo que definimos con una clase en el apartado 8.1. Dijimos que tendrían base y altura pero no les dimos un valor. Cuando creamos el objeto, es decir un triángulo dado, le vamos a asignar valores a estas propiedades.

Por ejemplo, partiendo de la clase Triangulo anterior vamos a definir dos objetos triángulo uno con la base 2unids, altura la 3unds y otro con la base 4unids, la altura 7unds.

Vemos cómo creamos diferentes objetos del tipo Triangulo. A estos objetos les pasamos diferentes valores.

```
Triangulo t1 = new Triangulo(2.0f,3.0f);
Triangulo t2 = new Triangulo(4.0f,7.0f);
```

Si queremos hallar el área de estos triángulos llamamos a su método definido anteriormente en la clase.

```
t1.area(); // Área 3.0
t2.area(); // Área 14.0
```

9. LA CLASE PRINCIPAL Y EL MÉTODO MAIN.

Un programa puede construirse empleando varias *clases*. En el caso más simple se utilizará una única clase. Esta clase contiene el programa, rutina o método

principal: main() y en éste se incluyen las sentencias del programa principal. Estas sentencias se separan entre sí por caracteres de punto y coma. Hay que tener en cuenta que en java se compone de ficheros .java. En cada fichero tendrá que ir una clase con el mismo nombre que el fichero .java y que deberá ser pública (más adelante profundizaremos en esto). Dentro del fichero podemos definir más clases, que tendrán otros nombres y no serán públicas.
La estructura de un programa simple en Java es la siguiente:

```
public class ClasePrincipal {
    public static void main(String[] args) {
        sentencia_1;
        sentencia_2;
        // ...
        sentencia_N;
    }
}
```

Como primer ejemplo sencillo de programa escrito en Java se va a utilizar uno que muestra un mensaje por la pantalla del ordenador. Por ejemplo, el programa Hola:

```
/**
 * La clase hola construye un programa que
 * muestra un mensaje en pantalla
 */
public class Hola {
    public static void main(String[] args) {
        System.out.println("Hola, ");
        System.out.println("me llamo Marga");
        System.out.println("Hasta luego");
    }
}
```

Como se ha indicado anteriormente, en un programa de Java todo se organiza dentro de las *clases*. En el ejemplo anterior, Hola es el nombre de la clase principal y del archivo que contiene el código fuente, Hola.java.
Todos los programas o aplicaciones independientes escritas en Java tienen un método main o principal que, a su vez, contiene un conjunto de sentencias, los conjuntos o bloques de sentencias se indican entre llaves { }. En el caso anterior, el conjunto de sentencias se reduce a tres sentencias, que son llamadas a dos métodos predefinidos en Java (print y println) que permiten visualizar texto por el dispositivo de salida de datos por defecto (la pantalla).

10. OPERACIONES DE ENTRADA/SALIDA.

En Java, **los flujos (streams)** son abstracciones que permiten leer o escribir datos de forma secuencial. Los flujos son útiles para manejar entradas y salidas de datos, ya sea desde archivos, teclado, redes, o cualquier otra fuente de información. La idea de un flujo es que los datos "fluyen" de un lugar a otro, facilitando la manipulación y transporte de información dentro de un programa.

10.1 SALIDA ESTÁNDAR.

La salida estándar será la pantalla y usaremos el flujo de salida System.out. Si queremos que algo se nos muestre podemos usar siguientes líneas de código:

```
System.out.metodo();
```

donde el método puede ser cualquiera de la tabla 11:

MÉTODO	DESCRIPCIÓN
println()	Imprime en una línea lo que va dentro del paréntesis. Puede ser cualquiera de los datos primitivos, arrays u objetos.
print()	En este caso imprime la información sin hacer el salto de línea.
printf("cadena de formato", variables)	Permite usar una cadena con variables sin necesidad de utilizar "+", para ello recibe un número variable de parámetros. El primer parámetro es fijo y es la cadena de formato. En ella se incluye texto a imprimir literalmente y marcas a reemplazar por texto que se obtiene de los parámetros adicionales. No hace salto de línea.

Tabla 11.

Los valores especiales que se pueden usar con printf(), son los siguientes:

CARÁCTERES ESPECIALES	SIGNIFICADO
%d	Entero.
%c	Carácter.
%s	Cadena.
%f	Float o double
%o	Entero en octal
%h	Entero en hexadecimal
%n	Salto de línea

Tabla 12.

La gran ventaja que presenta el printf es que se puede controlar el formato con el que imprimimos las cosas. Veamos algunos ejemplos:

```
double cantidad = 71283.567811;
entero cantidad2 = 75;
```

- Formato normal double y float:

```
System.out.printf("El valor de la variable cantidad es %f", cantidad);
```

- Formato con 2 decimales `%.2f`:

```
System.out.printf("El valor de la variable cantidad es %.2f", cantidad);
```

- Formato con símbolo + y 2 decimales `%+.2f`:

```
System.out.printf("El valor de la variable cantidad es %+.2f", cantidad);
```

- Formato separando los miles (71,283.567811 en lugar de 71283.567811)

```
System.out.printf("%nEl valor de la variable cantidad es %,f", cantidad);
```

- Formato reservando 5 huecos para un valor, útil para alinear. %5d

```
System.out.printf("%nEl valor de la variable cantidad es %5d", cantidad2);
```

```
   75  (tengo 3 espacios antes del número porque ocupa 5)
```

10.2 ENTRADA ESTÁNDAR.

Para la entrada estándar usaremos el flujo de entrada System.in que será lo que escribamos en el teclado y usaremos la siguiente sintaxis:

```
System.in.read();
```

Sytem.in solo tiene el método read que sirve para leer el entero equivalente al código ASCII del carácter introducido por teclado. Nota: si quisiéramos mostrar ese carácter como tal por pantalla habría que volverlo a pasar a código ASCII.

```
int num=System.in.read();
System.out.printf("%c", (char)num);
```

Como esta clase es bastante limitada hay otras que podemos usar:

- **BufferedReader**: únicamente posee el método *readLine()* para leer la entrada y este siempre retorna String. Para usarlo es necesario crear un objeto BufferedReader, así como lanzar la excepción en el main e incluir las librerías.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public static void main(String[] args)throws IOException {
      /*creamos el objeto de la clase BufferedReader, lo
      llamamos br */
      BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in));

      /*usamos ese objeto con el método readLine para leer
      una línea completa que la asignaremos a un String*/
      String cadena =br.readLine();
}
```

- **Scanner**: en este caso el objeto posee diferentes métodos para leer diferentes datos, ver tabla 13. Necesitaremos la librería **java.util**.

```
import java.util.Scanner;

public static void main(String[] args)  {

      /*Creamos el objeto lector de la clase Scanner, lo
      llamamos sc */
      Scanner sc = new Scanner(System.in);

      /*Asignamos la entrada a x, dependiendo del método que
      usemos x deberá ser definido de un tipo u otro. */
```

```
        x = sc.metodo();
    }
```

MÉTODO	DESCRIPCIÓN
nextByte()	para leer un dato de byte.
nextShort()	para leer un dato de short.
nextInt()	para leer un dato de tipo int.
nextLong()	para leer un dato de tipo long.
nextFloat()	para leer un dato de tipo float.
nextDouble()	para leer un dato de tipo double.
nextBoolean()	para leer un dato de tipo boolean.
nextLine()	para leer un String hasta encontrar un salto de línea.
next()	para leer un String hasta el primer delimitador, generalmente hasta un espacio en blanco o hasta un salto de línea.

Tabla 13.

Las diferencias entre ambos objetos son:

1. BufferedReader tiene una memoria buffer más grande (8KB), Scanner tiene una memoria buffer más pequeña (1KB).
2. BufferedReader es más rápido. Scanner es más lento en la ejecución ya que BufferedReader solo lee datos, pero el escáner también analiza datos.