TEMARIO PROGRAMACIÓN

Código Java

Código Html

TEMA 1

Unidad 1: Introducción a la programación

Lenguajes de alto nivel.

Lenguaje de bajo nivel:

Entiende el Hardware específico, de tendría que escribir un programa específico para cada procesador y cada componente del ordenador.

Hoy en día los lenguajes de bajo nivel sólo se usan para programar instrucciones de muy alto rendimiento, pues son donde un buen programador/a puede aprovechar mejor el hardware.

Lenguaje de alto nivel:

Los lenguajes de alto nivel solucionan los 2 problemas de los lenguajes de bajo nivel:

- Son fáciles de entender para el ser humano.
- Son independientes del hardware, son compatibles para cualquier Hardware.

Se pueden clasificar en 2 tipos, Compilados o interpretados.

Estos lenguajes de bajo o alto nivel siempre están escritos en archivos de texto. En estos lenguajes, hay instrucciones que se ejecutan **secuencialmente**: Una detrás de otra.
Una **instrucción** es una orden simple que se le da al programa.

1.1. Conceptos básicos.

1. Programa: Secuencia de instrucciones escritas en código que realizan una función.
2. Código fuente: Lo componen uno o varios ficheros de texto, escritos en un lenguaje de programación determinado, que dictan cómo se ejecuta un programa.
3. Compilar: Traducir de lenguaje de alto nivel a lenguaje máquina o de bajo nivel.
4. Compilador: Programa que se encarga de recibir un código fuente, y generar el código en lenguaje de bajo nivel que le corresponde.
5. Interpretar traducir en tiempo de ejecución de un programa una o varias instrucciones de lenguaje de alto nivel, y convertirlas en lenguaje de bajo nivel.
6. Intérprete: Programa que se encarga de interpretar código.
7. Memoria (o memoria principal) : La memoria RAM
8. Almacenamiento secundario: Almacenamiento permanente de la información. Suele referirse al disco duro, aunque puede ser cualquier otro medio permanente conectado al sistema.
9. Palabra reservada: Una palabra que ya se usa para algo en el lenguaje de programación, y no la podemos usar para ninguna otra cosa.
10. Entrada de un programa: Conjunto de datos o flujo que se introducen en un programa. Se introducen mediante dispositivos de entrada. El más obvio, un teclado.
11. Salida de un programa: Conjunto de datos o flujo que el programa emite hacia un flujo o dispositivo de salida, el más obvio, el monitor.

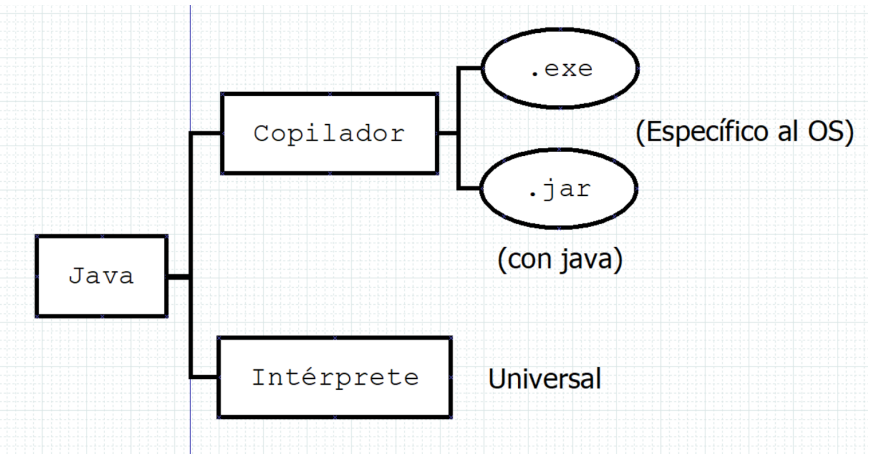
1.2. Lenguajes compilados e interpretados.

- Los lenguajes compilados son aquellos que para ser traducidos a lenguaje máquina, pasan por un compilador que traduce todo el código fuente de una vez. Si hay un cambio en el código, se debe volver a compilar el código entero. El resultado de pasar por el compilador es un ejecutable.
- Los lenguajes interpretados son aquellos que utilizan un intérprete para traducir el código fuente a lenguaje máquina. No generan un ejecutable, sino que el intérprete le va pasando al SO línea a línea lo que necesita.

Compilador / Intérprete (traducir código a binario).
-Leer y copila ficheros a .exe (ejecutable) o .jar
-No genera archivos. Traduce y ejecuta cada línea por separado en secuencia.

Los compiladores son más rápidos que los intérpretes.
Los .exe son específicos al sistema operativo, uno de windows no te vale para un Linux.
En el caso de un intérprete si te es válido.

Java tiene las 2 funcionalidades: un compilador para los programadores y el intérprete para los usuarios.



1.3. Variables y Constantes.

Son datos que utiliza el programa para trabajar, se almacenan en memoria principal, y se les da un nombre. En la mayoría de lenguajes de programación el programa necesita que se le diga qué tipo de dato va a haber en la variable.

Con las variables se pueden hacer varias cosas:

- Declararlas
- Inicializarlas
- Consultarlas
- Modificarlas
- Liberarlas

- Declarar una variable consiste en indicarle su existencia, y reservar su nombre para el programa mientras éste se ejecuta. Es como decirle al programa: "Acuérdate de que tienes que guardar un número, y lo vas a recordar por el nombre n1"

- Inicializar una variable consiste en asignarle un valor por primera vez. Suele ir siempre junto a la declaración, aunque no es forzoso. Es como añadirle a lo anterior: "... y n1 vale diez".

- Consultar una variable es recuperar el valor a través del nombre. Como decirle al programa: "Dime el valor de n1"

- Modificar el valor de la variable. Hay varias formas, lo más normal es usando el operador de asignación =.

- Liberar es hacer que dejen de ocupar espacio en memoria antes de que termine la ejecución del programa. Es como decirle al programa: "Olvida n1".

- Las constantes son variables con una propiedad especial: Una vez que se inicializan, no se les puede volver a cambiar el valor. Es decir, pierden la opción de modificarlas. Con ellas podemos:

  - Declararlas
  - Inicializarlas
  - Consultarlas
  - Liberarlas

¡Pero no podemos modificarlas!

CONCEPTOS

- Case sensitive

Sensible a mayúsculas, es decir: que no es lo mismo hola que Hola.

- lowerCamelCase

Todas las variables y nombres de funciones. vinoPodrido (muy importante para java).

- UpperCamelCase

VinoPodrido (clases de variables).

1.4. Tipos de datos y literales.

Las variables y constantes se deben declarar de un tipo de dato. Esto es necesario porque el programa tiene que saber cuánta memoria principal reservar para ese tipo de dato. El sistema operativo tiene que ser cuánto más conservador mejor asignando memoria, o se quedaría sin memoria rápidamente, y todo empezaría a ir lento.

Existen infinitos tipos de datos pero todos están compuestos de combinaciones de tipos primitivos (o básicos). Pueden variar un poco entre lenguajes, aunque en general, son siempre los mismos.

- Int (Integer) sirve para almacenar números enteros.
- Float sirve para almacenar números con decimales.
- Double sirve para almacenar números con decimales, pero permite números más grandes con más decimales.
- Char (Character) sirve para almacenar una sola letra o símbolo, que siempre va entre ‘ y ‘.
- String sirve para almacenar cadenas de texto hasta longitudes muy largas, que siempre van entre “ y “.
- Boolean sirve para almacenar valores verdadero (true) o falso (false).

Los literales son elementos que indican directamente (literalmente un valor).

- **Enteros:** usando el número.
  Ejemplo: int miVariable=-3; (3 es el literal).
- **Double:** usando un número.
  Ejemplo: double pi= 3.14; (3,14 es el literal).
- **Float:** usando un número seguido de una f.
  Ej: float pi= 3.14f; (3,14f es el literal).
- **Boolean:** con la palabra true o false.
  Ej: boolean estaVivo=true; (true es el literal).
- **Caracteres:** usando el caracter entre ‘ y ‘.
  Ejemplo: char letraA = ‘a’; (‘a’ es el literal).
- **String:** usando la cadena de texto entre “ y “.
  Ej: String saludo = “Hola”; (“Hola” es el literal).

1.5. Operadores y expresiones.

Operador de asignación, se escribe =. Asigna a la parte de la izquierda, donde normalmente pondremos una variable, el valor de la parte derecha.
Ejemplo:
Int edad=32;

Otros operadores muy comunes son: Suma + , Resta - , Multiplicación * , División / y Módulo %.
Funcionan de la misma forma: Con un operando a la izquierda y uno a la derecha, calculan el resultado de la operación. Ejemplos:
3+5;
nombreVariable + 9;
Este resultado se devuelve a la izquierda de la operación, y se puede asignar con el operador de asignación. Ejemplo:
nombreVariable = nombreVariable+9;

Desplazamiento a la izquierda <<
Que tiene un valor a la izquierda y un número a la derecha que indica cuántos bits de desplazamiento habrá, multiplicando el número por dos con cada desplazamiento. El resultado se devuelve a la izquierda de la operación.
Ejemplo:
variable = variable << 2;
Multiplica por cuatro (2*2) el valor de variable.
Desplazamiento a la derecha<<
Complementario al anterior, dividiendo el número por dos con cada desplazamiento. El resultado se devuelve a la izquierda de la operación.
Ejemplo:
variable = variable >> 2;
Divide por cuatro (2*2) el valor de variable.
Están también los operadores relacionales, que se usan igual que los aritméticos (+ - * / %), pero que lo que hacen es comparar dos valores y devuelven un valor booleano true o false.

==	igual a
!=	distinto de
>	mayor que
>=	mayor que o igual a
<	menor que
<=	menor que o igual a

2. La consola/terminal y los argumentos.

Una ruta es un camino para llegar a un archivo/fichero. Los distintos elementos que hay que atravesar para llegar a ese archivo/fichero destino se separan por / (linux) o \ (Windows). Tenemos dos modalidades de ruta:

- **Absoluta:** (C:X\x\...) Comienza desde el nombre de la unidad.Informa cómo llegar desde cualquier lugar.
  Ejemplo: C:\Usuarios\cenec\Documentos\...

- **Relativa:** (".\") Comienza desde la carpeta actual donde nos encontremos.Te indica dónde encontrar el archivo desde la misma carpeta. (solo sirven desde esa carpeta X).
  Ejemplo: ../NetBeansProjects

- **La carpeta .** referencia a la misma carpeta donde me encuentro.
- **La carpeta ..** referencia a la carpeta madre que contiene mi carpeta actual.

Ambos se pueden utilizar para construir rutas relativas.

CMD

- **cd -** Viene de change directory, sirve para cambiar directorio.(cambiar directorio).
- **cp -** Copia un archivo de una ruta a otra.
- **mv -** Mueve un archivo de una ruta a otra.
- **del** (o rm en PowerShell y Linux) **-** Borra un archivo.
- **mkdir -** (crear carpeta).
- **rmdir -** (eliminar carpeta).
- **cls -** (limpiar pantalla).
- **man -** … (manual).
- **ls -** (enseñar contenidos).

TEMA 2

Unidad 2: Introducción a la programación con Java y repositorios

1. Conceptos.

VERBOSE

Es un modo de salida de un programa, que se usa para dar información detallada de todo lo que hace el problema en todo momento. Genera mucha más salida de la que normalmente generaría el programa, y se suele usar para ayudar en la depuración.

ÁMBITO

Un Ámbito (Scope en inglés) es una región del código fuente, que se encuentra entre llaves ({ y }). En ella puede haber instrucciones u otros ámbitos. Un ámbito se abre con { , y se cierra con }. Siempre deben cerrarse, o el programa será erróneo.
Siempre que hay una llave de cierre, esta cierra el último ámbito que se abrió. Los ámbitos determinan, entre otras cosas, el lugar donde existen las variables. Si declaro una variable dentro de un ámbito, esta deja de existir cuando el ámbito se cierra.

CONSOLA

La Consola de Java es un lugar en el que tu programa se comunica contigo a través de texto. Allí va la salida estándar, y de ahí viene la entrada estándar. Sirve tanto como para que el usuario del programa lea, como para que el usuario del programa escriba.

2. Entornos de desarrollo.

Java Development Kit (JDK) es un conjunto de herramientas que nos permite desarrollar, compilar y ejecutar programas en Java. Hace ya muchos años que se incluye en el JSE (Java Standard Edition). Sus herramientas más importantes son:

- El compilador javac, con el que convertimos nuestro código en java, con extensión .java en código intermedio compilado (bytecode) con extensión (principalmente) .class. Estos .class se pueden agrupar en un .jar, que es un comprimido ejecutable.

- Javadoc, programa encargado de generar la documentación de nuestro código.

La máquina virtual de Java (JVM), es la herramienta encargada de interpretar el código intermedio (bytecode) y traducirlo de forma interpretada a lenguaje máquina, lo que convierte a java en un lenguaje semi-compilado y semi-interpretado. Está incluída dentro del JRE.

El Java Runtime Environment (JRE), contiene justamente la máquina virtual, está separada del JDK porque los usuarios finales que no van a programar no necesitan el JDK, y pueden obtener esta herramienta mínima.
El JDK siempre tiene en su interior el JRE.

La compilación Just In Time (JIT), es una característica por defecto habilitada de la JVM, por la que, en lugar de interpretar el bytecode de los .class, compila el código justo antes de ser ejecutado en código máquina.
De este modo, las instrucciones no se interpretan en la JVM, sino que se ejecutan ya compiladas al código de más bajo nivel, aumentando su eficiencia.

Los Entornos de Desarrollo Integrado (IDE) son conjuntos de software normalmente agrupados en una interfaz, que nos ofrecen las herramientas básicas que necesitamos para escribir y testear software. Normalmente contienen:
- Un editor de código.
- Un compilador/intérprete.
- Un debugger.

3. Bases de Java.

Java es un lenguaje semi-compilado y semi-interpretado, portable, orientado a objetos, concurrente y de propósito general.

- Necesitamos compilar con el compilador javac, incluído en el JDK, el código de alto nivel, .java, para poder ejecutarlo luego con el intérprete:
  > javac archivo.java [opciones]
- Podemos compilar de golpe una lista de archivos también:
  > javac archivo.java archivo2.java archivo3.java [opciones]
- O utilizar wildcards:
  > javac *.java [opciones]
- También podemos usar un archivo de texto donde en cada línea se indica un archivo .java a compilar, y compilarlos todos de golpe con:
  > javac @miarchivodetexto

En Java todo va dentro del ámbito de una clase: Incluso la función main, que es necesario que esté presente para que una JVM sepa que ahí tiene que iniciarse la ejecución de un programa.
Para construir un programa correctamente en java, debe contener al menos un main, la indentación debe ser correcta para facilitar la legibilidad del código, se deberían respetar espacios en blanco entre operadores, y saltos de línea simples entre instrucciones, y dobles al final de métodos y clases.

4. Operadores.

Un operador es un elemento del lenguaje que se aplica a variables, constantes o resultados de métodos, y permite modificar o comparar el valor de nuestras variables. Puede ser:

- **Usuarios:** (1 operando) si trabajan con un solo valor colocado a la derecha o izquierda del operador. EJ:
  !variable;
- **Binarios:** (2 operandos, izq y der.) si trabajan con dos valores, cada uno colocado a un lado del operador. EJ:
  variable+3

No todos los operadores tienen sentido con todos los tipos de datos: Hay algunos que significan cosas distintas para distintos tipos de datos. Ej: + es para suma si es con números o char, o para concatenar si es con String. Pero, no tiene sentido usar + con booleanos.
El uso de al menos un operador, constituye una expresión. Puede haber expresiones con un operador, o con un montón:
Es una expresión: variable+3
Es una expresión: variable+3-5*2/4

Las expresiones se evalúan (devuelven un valor), siempre en el orden de la precedencia de los operadores. A igual precedencia, se opera de izquierda a derecha. O dándoles precedencia con los paréntesis (3 + 5) * 2.

Orden de precedencia de los operadores.

1. **Acceso array** "**[ ]**"
2. **Método y miembros** "**.**"
3. **Paréntesis** "**( )**".
4. **PostIncremento. expr++** Primero suma el valor y después lo imprime.
5. **PostDecremento. expr–** Primero resta el valor y después lo imprime.
6. **PreIncremento.** Primero imprime el valor y después lo suma.
7. **PreDecremento.** Primero imprime el valor y después lo resta.
8. **Suma unaria *+expr.***
9. **Resta unaria *-expr.***
10. **Not lógico !** (Sólo para boolean).
11. **not a nivel de bit ~** (Solo para números enteros byte-long).
12. **Casting (tipo)**
13. **Creación de objeto new**
14. **Multiplicación, División y Módulo** "*****" "**/** " "**%**"
15. **Suma** (y concatenación) y resta "**+**" "**-**"
16. **Desplazamiento** (Solo para números enteros byte-long)"**<<**" "**>>**" "**>>>**"
17. **Relacionales** "**<**" "**>**" "**<=**" "**>=**" "**instanceof**"
18. **Igualdad** "**==**" "**!=**" "**.equals**". para comparar un valor numérico / texto.
19. **AND** a nivel de bits **&**
20. **XOR** a nivel de bits **^**
21. **OR** a nivel de bits **|**
22. **AND** lógico **&&.** Devuelve falso, a no ser que los 2 elementos a sus lados sean true.
23. **OR** lógico **||.** Devuelve verdadero, a no ser que los 2 elementos a sus lados sean falsos.
24. **IF ternario** "**?**" "**:**"
25. **Asignación** "**=**" "**+=**" "**-=**" "***=**" "**/=**" "**%=**" "**&=**" "**^=**" "**|=**" "**<<=**" " **>>=**" "**>>>=**"

5. Entrada y Salida.

Salida se refiere a lo que sale del programa (normalmente la pantalla), y entrada a lo que entra en el programa (normalmente el teclado).
De hecho la pantalla y el teclado se llaman respectivamente, la salida y la entrada estándar, porque un programa por defecto, está “enganchado” a ellas.
La salida más sencilla es usar los métodos print* de System.out.
Ej: System.out.println(Cualquier variable o expresión de cualquier tipo);

La entrada básica se hace con System.in.read(), que para el programa cuando llega a esa línea, hasta que se introduce algo por teclado: un carácter e intro.
Si queremos leer más de un carácter, tenemos que poner más de un System.in.read().
¡Cuidado! porque intro también es un carácter.

```
Ej:     char p0=(char)(System.in.read()+1);
                char p1=(char)(System.in.read()+1);
                char p2=(char)(System.in.read()+1);
```

Para facilitar la entrada, java tiene incorporada la clase Scanner. Esta clase permite leer una línea de una sola vez, y meterla en un String, int, float, boolean…
Para usarla, hay que declarar una variable de tipo Scanner:

```
Scanner sc;
```

La podemos inicializar así:

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

y leemos una línea con el método nextLine():

```
String leido=sc.nextLine();
```

Eso nos mete en un String una sola línea: todo lo que se haya escrito antes de pulsar intro.

6. Conversiones de datos.

Conversión o Casting, es la acción de indicar que quiero convertir un tipo de dato a otro con el que es compatible. No se puede hacer entre dos tipos de datos cualquiera, solo entre los que “se entiendan” como compatibles.
Para hacer un casting, delante del dato que se quiere convertir se pone entre paréntesis el tipo de dato al que queremos convertir.
Ej:

```
byte varByte=30;
byte resByte=(byte)(30+varByte);
```

En varByte meto un literal de entero. Los byte no tienen literal propio, así que necesitamos a los enteros para definir valores byte.

Si aquí no pusiera el (byte) , que es lo que indica el casting o conversión, java protesta, porque como no sabe lo que vale varByte, y el resultado podría ser más grande de 127, y no caber. Indicarle casting implica que yo como programador me responsabilizo de usar esos datos adecuadamente, y de que quepan. Cuando queremos hacer casting a una expresión, tenemos que ponerla entre paréntesis.

Si me paso sumando dos bytes, va a suceder una cosa que se llama overflow (desbordamiento) , que quiere decir que los bytes que sumas, no caben en un solo byte, y entonces podrías internamente cambiar el signo y el significado del número.

COMPATIBILIDADES
Tipos enteros, con decimales y char son compatibles, siempre que se usen con responsabilidad, a través de casting estamos haciendo lo mismo.

Entre float e integer, que tienen el mismo tamaño, por velocidad es por lo que se usa int.
Puedo meter un entero en un float.

```
float f=3;
```

7. Comentarios y organización.

Los Comentarios son textos que hay en el código fuente de ayuda al programador. Java (y cualquier otro lenguaje) los ignora.
Hay dos tipos de comentarios en java, uno sirve para comentar en una sola línea, y otro para comentar en múltiples.

Entre **/*** y ***/** puedo poner comentarios de todas las líneas que quiera. Si dentro hay código, se ignora.
Lo que haya detrás de **//** es un comentario hasta que acabe la línea. La siguiente línea se entiende que vuelve a ser código. Si se quiere comentar una sola línea de código, estos comentarios se ponen o justo encima o a la derecha de la instrucción en concreto.
Ejemplos:

```
/* esto es un comentario en varias
líneas, java lo ignorará por completo, solo sirve para ayudar al
programador */

//Este es un comentario en una sola línea. No puede ocupar más.
```

Un programa en Java para funcionar, no necesita más que el código fuente escrito en (al menos) un fichero .java, que se compile en un .class y un .jar opcionalmente, y se ejecute en el intérprete.

Los **IDE** nos ayudan a manejar nuestro código fuente de forma más ordenada, a través de proyectos. Los proyectos son estructuras de carpetas, que contienen el código fuente, el compilado en class, y otros ficheros que el IDE necesita para organizarse.

Organización en los ficheros.

- **src.** Es donde está nuestro código fuente (source). Tendrá dentro las carpetas que correspondan al paquete o paquetes donde esté el código.

- **build / bin.** Es la carpeta donde, bajo la misma estructura que corresponde a los paquetes, los .class compilados.

- **dist.** Contiene los .jar generados de la compilación y el empaquetado del programa: Ejecutables listos para el intérprete. Esta carpeta no está en eclipse: Exporta a jar de otra forma.

TEMA 3

Unidad 3: Estructuras de control y flujo

Apuntes dados del Profe

Apuntes condicionales y bucles.docx

Contenido

1. Conceptos.

Random

La clase Random, permite generar valores pseudoaleatorios de cualquier tipo de dato básico. La función nextInt saca un aleatorio entre cero (incluído), y el número que le pongamos entre paréntesis(sin incluir). Este ejemplo saca un nº entero aleatorio entre 0 y 24.

```
int rand=aleatorio.nextInt(25);
```

El Random se tiene que importar como el Scanner, se hace de la siguiente manera:

```
Random rand = new Random ();        //Siendo "rand" el nombre que le pondremos.
```

Si quiero sacar un aleatorio en un intervalo se hace con:
valor = inicialRango + aleatorio.nextInt(final-inicial). **Ej:**

```
int rand = 5 + aleatorio.nextInt(10 - 5); //Aleatorios entre 5 y 10.
```

Este valor se saca del Epoch (fecha) 1 enero 1970 a las 00:00.

2. Estructuras de selección.

Normalmente un programa no va a ejecutar todas las instrucciones en orden de la primera a la última: Hay decisiones que puede tomar el usuario o el programa.
La forma más simple, y la más común de establecer estas decisiones en código es con la instrucción if.

IF / ELSE

Los ifs se montan así:

```
if (expresión booleana) {
        //lo que ocurre si la condición se cumple. De 1 a virtualmente
        //infinitas instrucciones
}
```

-if es una palabra reservada
-entre paréntesis va una expresión que devuelve true o false
-después de los paréntesis, un ámbito que indica lo que pasa si la condición se cumple.

Además, los lenguajes de programación nos ofrecen también el ámbito else, que se ejecuta solo si no se cumple la condición del if. Así, en un if con else, siempre se ejecuta o solamente el ámbito del if si la condición se cumple, o solo el ámbito del "else" si no se cumple. La palabra reservada "else" siempre se pone detrás del cierre del ámbito del if. No puede haber un else sin if.

Ejemplo:

```
public static void main(String[] args){
  Scanner sc=new Scanner(System.in);
        System.out.println("Dime tu edad");
        byte edad=Byte.parseByte(sc.nextLine());
        if (edad>=18) { //Solo se ejecuta lo que hay aquí si la edad es más de 18 0 18.
                System.out.println("Eres mayor de edad. Bébete una cerveza.");
                System.out.println("Bien por ti.");
    } else {  //Solo se ejecuta lo que hay aquí si la edad es menos de 18.
                        System.out.println("Eres menor de edad. Bébete una fanta.");
            }
         System.out.println("No tengo más preguntas");
```

Si se ha introducido una edad mayor o igual a 18, la condición del if se cumple, y el programa entra a ejecutar las instrucciones de su ámbito. Si no, continúa por la siguiente instrucción al cierre del ámbito. Podemos poner tantas instrucciones dentro del ámbito del if como queramos.

Dentro de un if, yo puedo poner otro if, sin límite ninguno de veces. Un if dentro de otro, se llama if anidado.
En ifs que van uno detrás de otro podemos reducir el número de comprobaciones en algunas situaciones para que el programa vaya un poco más rápido. **Por ejemplo:**
if { } else if { } else if { } else …
En ese caso, la instrucción que está detrás del else sin ámbito es un if, y todo su ámbito y el del else actúan como una sola instrucción: Si se cumple la condición del primer if, el segundo, que está en el else sin ámbito ni se mira. Si no se cumple, si se comprueba la condición del segundo if.

Importante: Las variables solo existen dentro del ámbito donde se declaran, y en sus ámbitos interiores.
Por lo tanto, si lo metemos en un ámbito o bucle (if / while / switch / funciones, etc), no se podrán usar fuera de este de este.

IF TERNARIO

La mayoría de lenguajes de alto nivel, nos proporcionan el operador ternario (?:). Es ternario porque tiene 3 operandos, y es una abreviación de if para casos de asignación. **Se usa así:**

(Expresión booleana ? Instrucción si true : Instrucción si false)

En Java se usa distinto: no lleva paréntesis, y tiene que devolver algo, por tanto después de **?** y de **:** hay que poner el valor que se asigna.

Expresión booleana **?** valor si true **:** valor si false

Ej:

```
System.out.println( edad>17?
            "Eres mayor de edad: Op. Ternario":
            "Eres menor de edad: Op. Ternario");
```

SWITCH

Los lenguajes de alto nivel nos dan la instrucción switch, para hacer estos casos escribiendo menos. No es más rápido ni lento que encadenar ifs, solo es una manera más fácil de escribirlos. En cada (case … break;) expresa lo que se hace en el caso que la expresión devuelva el valor que pone el case. Dentro de un switch, no se pueden repetir los literales en los case. No podría poner dos veces case 'a'.

Además de los case, podemos poner en último lugar un default, para todos los casos que no se contemplan en el case. El "default" es usado en el caso de que ninguna de los cases se cumplan. Se puede poner entre cualquier case...break, pero se suele poner el último para no confundir. **Su sintaxis es:**

```
switch (expresión que devuelva byte,short,int,char o String) {
        case literalValorPosible:
                //instrucciones que ocurren si la expresión devuelve
"literalValorPosible"
        break;
        case literalValorPosible2:
                //instrucciones que ocurren si la expresión devuelve
"literalValorPosible2"
        break;
        …
        default:
            System.out.println("No sé por qué letra empieza tu nombre");
          break;
}
```

Break en realidad no tiene por qué estar, lo que hace es parar la ejecución de instrucciones cuando se ha entrado en un case.
El case sirve como "if", si pasa esto, lo siguiente.
Entre el case y el break se localiza la acción si se cumple el case.

3. Bucles.

Los bucles nos sirven para hacer cosas varias veces, sin tener que repetir el código varias veces. Dar una vuelta por todas las instrucciones que pertenecen al ámbito de un bucle se llama iterar. Cada vuelta es una iteración. Hay tres tipos de bucles:

1. **while:** Se usan cuando no sabemos cuántas veces lo vamos a repetir. Podría no ejecutarse ni una vez.
2. **do...while:** Se usan cuando no sabemos cuántas veces lo vamos a repetir, pero queremos que se repita al menos una.
3. **for:** Sirven para repetir instrucciones un número determinado de veces, cuando conocemos de antemano cuántas veces lo vamos a repetir.

WHILE

Los bucles while son útiles para cuándo no sabemos cuántas iteraciones vamos a hacer. Su estructura y funcionamiento es así:

```
1 -  inicialización de variable iterador (Fuera del bucle - Opcional)
2 - condición de parada (expresión booleana)
3 - Iteración en el ámbito
4 - Incremento del iterador (Opcional)

int numero=0;
int i=0;
while(numero!=10){
 System.out.println("Intento : “+i+” Escribe el numero 10:");
            int numero=Integer.parseInt(s.nextLine());
            if(numero==10){
               System.out.println("Bien hecho");
                                    }
                                    i++;
```

No tenemos por qué tener un iterador, porque no sabemos cuántas veces vamos a pasar por el bucle. Se puede usar para comprobar que tenemos datos de entrada correctos. **Por ejemplo, la edad:**

```
System.out.println("Dime tu edad:");
      Scanner sc = new Scanner(System.in);
      short edad = Short.parseShort(sc.nextLine());
      //Me aseguro de que no introduzca un número negativo ni mayor a 127.
      while (edad <= 0 || edad > 127) {
         System.out.println("Has metido la edad: "
               + edad + ". No es válida. Tiene que valer entre 1 y 127."
               + "Introdúcela de nuevo.");
         edad = Short.parseShort(sc.nextLine());
      }
      System.out.println("Correcto. Tu edad es: " + edad);
```

DO WHILE

Los bucles do…while son útiles para cuándo no sabemos cuántas iteraciones vamos a hacer, pero al menos queremos asegurar que se hace una. Nos conviene cuando no necesitamos iterador, para escribir la operación con menos líneas de código que si tuviésemos un for. Su estructura y funcionamiento es así:
1 - Inicialización de variable iterador (Fuera del bucle - Opcional).
2 - Palabra reservada do.
3 - Iteración en el ámbito.
4 - Incremento del iterador (Opcional).
5- Condición de parada (expresión booleana).

```
do {
 System.out.println ("Intento : “ + i + ” Escribe el numero 10:");
            int numero = Integer.parseInt (s.nextLine());
            if (numero == 10) {
               System.out.println ("Bien hecho");
                                    }
                                    i++;
} while (expresión booleana);
```

Break

La palabra reservada break sirve dentro de un bucle(de cualquier tipo) para parar el bucle, y salir de él apenas se llegue a break;. Esto se puede usar en situaciones en las que con que se cumpla una situación una vez, no hace falta seguir iterando.
El “break” sirve como la llave de cierre “}”, cierra la acción realizada.

Continue

Hay otra palabra reservada, continue, que te permite saltar el resto de la iteración en un bucle, y pasar a la iteración siguiente directamente. Apenas hay situaciones en las que nos facilite la vida, pues suele haber una forma más comprensible de reescribir el bucle en lugar de usarla. Por ejemplo, este bucle:

```
for (int i = 0; i < texto.length(); i++) {
          if (texto.charAt(i)=='a') {
             letraPresente=true;
                        System.out.println("La letra a está en la posición "+i);
          }
      }
```

Se puede escribir usando continue como:

```
for (int i = 0; i < texto.length() ; i++) {
          if (texto.charAt(i)!='a') {
             continue;
          }
                letraPresente = true;
          System.out.println ("La letra a está en la posición "+i);
      }
```

FOR

Los bucles for sirven para repetir instrucciones un número determinado de veces.
Su estructura es así:

for (“tipo variable” “comienzo” ; “final” ; “orden”).

```
for ( byte i = 0 ; i <= 10 ; i++ ) {
        System.out.println (numero + " x " + i + " = " + numero * i);
        }
```

```
System.out.println(“Linea normal 1”);
for ( int iter=0   ;  iter<10    ;  iter++  ) {
        System.out.println (“Iterando por el bucle”);
        System.out.println (“Haciendo pruebas”);
}
System.out.println (“Linea normal 2”);
```

Y sus partes se ejecutan en este orden:
1 - Inicialización de variable (Iterador),puede declararse en esta línea o estarlo antes (Solo en la 1º Iteración).
2 - Condición de parada (expresión booleana).
3 - Iteración en el ámbito.
4 - Incremento del iterador.

```
int i;
    for ( i = 0 ; i < 10 ; i = i+1) {
        System.out.println ("Iteración "+i);
    }
    System.out.println ("He salido del bucle");
```

En la 1º iteración, se inicializa i=0 (Paso 1).
Luego se comprueba que i<10 (Paso 2), y se ejecuta el sysout del ámbito (Paso 3).
Después se incrementa con i=i+1 (Paso 4), se vuelve a comprobar la condición (Paso 2), y si se cumple, se vuelve a ejecutar el ámbito (Paso 3).
Se repiten los pasos 2,3 y 4 hasta que la condición no se cumpla.

El comienzo establece desde que valor empezará a contar el bucle.
El final establecerá su final.
(Empezamos a contar desde el 0 hasta el 10, pasando de 1 en uno).
El orden establecerá como contará el for. desde “variable” sumándole o restándole 1 a 1.

4. Funciones.

Las funciones sirven para hacer mi código más pequeño, no repitiendo código que utilizo varias veces en el programa. Además, tiene la ventaja de que al escribir las cosas solo una vez, si hay un error en el código, solo lo tengo que arreglar una vez.

Funciones y procedimientos son sinónimos. Método es el nombre que se le da a una función dentro de una clase. En java solo existen métodos. Aunque se hable de funciones, son siempre métodos.
Una función se puede declarar (Especificar lo que devuelve, como se llama y lo que hace) o llamar (invocar) (Hacer que la función se ejecute).

Para declarar una función:
1 - Se ponen fuera del main, y dentro de la clase.
2 - Lo primero que se pone es public, private o protected.
3 - A nuestro nivel, de momento la segunda palabra va a ser siempre static.
4 - Después va el tipo de dato que se devuelve o void si no devuelvo ninguno.
5- Después va el nombre de la función **EN lowerCamelCase.**
6 - Paréntesis que pueden contener o no argumentos.
7 - ámbito con las instrucciones de la función.

Para invocar una función, se pone en el lugar que se quiera llamar, el nombre, y los paréntesis con los argumentos si los tiene, y punto y coma.
Dentro de los paréntesis de la declaración de la función pueden ir o no argumentos.

Los argumentos declaran variables que le “prometes” pasar a la función en su llamada. Sirven para pasar valores que están fuera de la función hacia dentro de la función

Se declaran en la declaración de la función, y se declaran como variables normales (tipo+nombre variable) separado por coma.
En la llamada, tenemos que meter tantos datos como le hayamos prometido en la declaración, y tienen que ser del mismo tipo en el mismo orden.

Cuando una función ejecuta una instrucción return, sale inmediatamente, devolviendo el valor indicado. No se ejecuta ninguna instrucción después del return.

Dos funciones son diferentes si tienen distinto nombre, o teniendo el mismo nombre si los atributos son distintos (distinto número, o distinto tipo, o distinto orden).

Ejemplo sencillo:
```
public class PruebaFunciones {
    public static void main(String[] args) {
        String algo = "aaaa";
        PruebaFunciones.suma (2,3);
    }

    public static void suma (int numero1,int numero2) {
        System.out.println (numero1 + "+" + numero2 + "= " + (numero1+numero2));
    }
}
```

En este ejemplo, fuera del main he declarado la función. Es decir: le he dicho el tipo que devuelve (void - nada), el nombre y entre paréntesis los argumentos que tiene. Después el ámbito y todas las instrucciones que hace al ser invocada.

Se invoca desde el main llamando a nombredeclase.nombredefuncion(argumentos);
Si la función está declarada dentro de la misma clase, puede invocar sin usar el nombre de clase:

```
public static void main(String[] args) {
        String algo = "aaaa";
        suma (2,3);
    }
```

Las funciones se distinguen por nombre, número de argumentos y tipo. No puedo poner dos funciones que se llamen igual con los mismos argumentos en el mismo orden, pero si dos funciones que se llamen igual con distintos argumentos o distinto orden. Con distintos argumentos, aunque se llamen igual, la invocación sabe cuál de las dos usar, porque le paso distinto número de argumentos, y ejecuta aquella en la que el número y tipo de argumentos coincide exactamente con los declarados:

```
suma (2,3);
```

o

```
suma (2,3,1);
```

No es buena idea escribir con System.out directamente dentro de las funciones (a no ser que sea para depurar), porque cuando tengamos muchas, va a ser difícil localizar dónde estaba el sysout.

Es mejor dejar los sout para el main, y ya tenemos todos localizados. Esto lo podemos lograr devolviendo un String cuando la función finaliza.

Esto se hace en tres pasos:

1 - Detrás de public static, en la declaración de la función, se pone el tipo de dato que prometemos que vamos a devolver en la función (String).
2 - Al principio de la función, declaramos un String inicializado a la cadena vacía. Al final de la función, devolvemos ese String.
3 - Todos los sysout los transformamos a concatenaciones en esa variable String.

Entonces, en el ejemplo de la tabla, o en cualquiera donde estemos imprimiendo algo por pantalla dentro de una función, podemos quitar los sysout de dentro de la función concatenando todo en un string dentro de la función, y haciendo sout a la invocación. **Ej:**

```
public static String tablaDel(byte num){
    String res = "";
    for (int i = 0; i <= 10; i++) {
        res += num + " x " + i + " = " + (num*i) + "\n";
    }
    return res;
}
```

Desde main, la llamada a la función, la podemos tratar como un entero, y utilizarla como tal. Por ejemplo, para asignar lo que devuelve a una variable.
O puedo utilizar el valor devuelto de una función como argumento de otra.
Los valores al salir de la función se pierden, porque al entrar, los valores de los argumentos se copian en las variables de los argumentos de la función, que existen solo en su ámbito. Cuando acabe la función, el ámbito de la función se borra, y solo nos queda lo que teníamos en Main.

Se puede hacer aún más corta, porque se pueden poner varios return dentro de una función. En cuanto la ejecución llegue a uno de ellos, la función no sigue, y devuelve ese valor. No se puede poner nada inmediatamente detrás de un return, porque la línea nunca se ejecuta.

```
public static boolean esMayorDeEdad (byte edad) {
        if (edad >= 18) {
            return true;
        }
        return false;
    }
```

Si hacemos esto, tenemos que tener cuidado de hacer un “return” en cada caso de ejecución posible, porque si no, hay situaciones en las que nunca se devuelve nada. Esto nos dá un error de compilación, como en este caso:

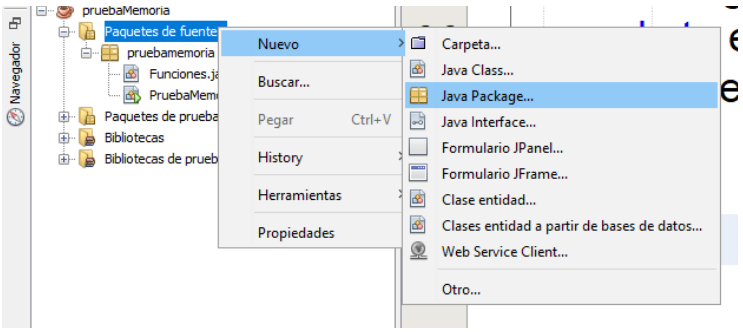
```
public static boolean esMayorDeEdad (byte edad) {
        if (edad >= 18) {
            return true;
        }
    }
```

Entonces, otra de las cosas que podemos hacer es llamar a una función dentro de otra. El número de funciones que podamos llamar dentro de otras no tiene límite.

```
public class Funciones {

    public static String pideDatoString(String mens
        Scanner scan=new Scanner(System.in);
        System.out.println(mensaje);
        return scan.nextLine();
    }

    public static String informe
        (String nombre,byte edad,String nacionalidad
            return nombre+", de "+edad
                +" años, y nacionalidad "+nacionalidad+
        }


    public static byte pideEdad(){
        Scanner scan=new Scanner(System.in);
        System.out.println("Dime tu edad");
```

Deberíamos meter todas las funciones en una clase aparte para tener más claro y visible todas las funciones que usamos.

No hay porqué meter el archivo de funciones dentro del mismo paquete. Puedo meterlo dentro de otro. Aunque siempre deberíamos evitar usar el default package. En ese caso, tengo que hacer import desde el archivo que quiera usar las funciones.



Sobre paquetes: Recordad no usar ninguna mayúscula ni número en su nombre, y que los paquetes son solo carpetas en el sistema de archivos, y que nos valen para organizar código, van a ser muy útiles cuando empecemos a tener muchos archivos distintos de código, para organizarnos nosotros mismos.

4.1. Recursividad.

Definición formal, y un poco infumable:
La recursividad es una forma de definir una secuencia de objetos, como una expresión, función o conjunto, donde se da un número de objetos iniciales, y cada objeto se define en función de los anteriores.

En programación, la recursividad es una propiedad que pueden tener las funciones: una función es recursiva si en su definición (implementación) se llama a sí misma. Tienen casi siempre la misma estructura: Comprueban una condición de parada (caso base), y si no se cumple, se llaman a sí mismas (caso recursivo).
Lo que se puede hacer con una función recursiva, se puede hacer de forma iterativa también a través de uno o varios bucles.

Bucle Main:

```
public static void main (String[] args) {
        Scanner sc = new Scanner (System.in);
        String respuesta = "";
        while (!respuesta.equals("si")) {
            System.out.println ("¿Comprendes los bucles while?");
            respuesta = sc.nextLine();
        }
        funcionRecursiva ();
    }
```

Este bucle preguntará, las veces que sea necesario, hasta que digas “si”, si comprendes los bucles while.

Bucle en Función:

```
public static void recursiva() {
        Scanner scan = new Scanner (System.in);
        String respuesta = "";
        System.out.println ("¿Comprendes la recursividad?");
        respuesta = scan.nextLine();
        if (!respuesta.equals("si")) { //Caso recursivo
            Funciones.recursiva();
        } else {  //Caso base
            System.out.println("Olé tu");
        }
    }
```

Es una función recursiva porque dentro de su propia definición se llama a sí misma en un caso (Caso recursivo) , y no lo hace en otro (caso base).

Son equivalentes. En su ejecución, la función recursiva va a ocupar más memoria, por todos los ámbitos que crea, pero esto solo dura un momento, por lo que acaba siendo irrelevante. Ya que esa memoria se libera conforme se destruyen ámbitos.

¿Cómo se pasa de un bucle a una función recursiva?

Da igual que sea un bucle for, while o do..while. Los pasos son siempre los mismos pasos, solo cambia que los while y do..while no tienen iterador, y se salta el paso 1:

1. Pasar por argumentos el contador (iterador)
2. Pasar cualquier otra variable que intervenga en la condición de parada por argumento
3. Normalmente (depende del programa) se pone el cuerpo del bucle lo primero
4. La condición para distinguir entre caso base y caso recursivo es la misma que la condición de parada del bucle.
5. Donde la condición de parada da true, se hace la llamada recursiva, sumando 1 al iterador
6. Donde la condición de parada no se cumple, si es necesario se hace algo.
7. En la llamada, el argumento se inicializa al mismo número al que se inicializa el iterador en el bucle.

Ejemplo:

```
    public static void main (String[] args) {
        Scanner sc = new Scanner (System.in);
        System.out.println ("Dime un numero");
        byte n1 = Byte.parseByte (sc.nextLine());
            //La función recursiva es equivalente a este bucle debajo.
        /*byte n2;
        do {
            System.out.println ("Dime otro (tiene que ser mayor");
            n2 = Byte.parseByte (sc.nextLine());
    } while (n2 <= n1);*/
    byte n2 = pedirNumeroMayor (n1);
}
```

Otro Ejemplo:

```
public class Funciones {
        public static void EntiendeRecursividad () {
                Scanner sc=new Scanner(System.in);
                System.out.println("¿Entiendes la recursividad?");
                boolean loEntiende = (sc.nextLine().toLowerCase().charAt(0)=='s');

                if (!loEntiende) { // casoRecursivo
                        Funciones.EntiendeRecursividad();
                } else { // casoBase
                        System.out.println("Me alegro!");
                } // If
        } // EntiendeRecursividad

} // Funciones
```

5. Errores más comunes.

If con = en lugar de ==.
Switch sin break.
Olvidar que se puede incrementar el contador en más de 1 en los bucles.
Hacer bucles infinitos.

Confundir contadores en bucles anidados.
Declarar funciones con mayúscula y confundirse.
Hacer funciones que no sean independientes de otras, o que intenten hacer más de una cosa.
Poner ; antes de abrir los ámbitos de switch,ifs,for,while…

6. Buenas prácticas.

Meter las funciones en ficheros aparte:
Crear nuevo -> Java class y en una clase sin función main, crear nuestras funciones, luego podemos usarlas desde cualquier otro fichero.
Si la clase está en otro paquete, hace falta indicarle import:

```
import nombrepaquete.NombreClase
```

La clase puede estar debajo de varios paquetes anidados, o en paquetes con punto en el nombre (por debajo se convierten en carpetas):

```
import paquete.paquete2.NombreClase;
```

Se pueden importar funciones sueltas:

```
import paquete.paquete2.NombreClase.nombreFunc;
```

Se pueden importar todas las funciones de una clase:

```
import paquete.paquete2.NombreClase;
import paquete.paquete2.NombreClase.*;
```

Si la función es estática, hay que ponerle static delante al import:

```
import static  paquete.paquete2.NombreClase.nombreFunc;
```

O todas las clases de un paquete:

```
import paquete.paquete2.*;
```

7. Depuración.

Depurar es corregir errores (bugs) en el código. En inglés se le llama debugging.
Hay varias técnicas para hacerlo, ahora mismo tenemos a nuestro alcance dos:

- Mostrar por pantalla información en momentos determinados
- Utilizar Breakpoints , y el debugger de Java.

Breakpoint es un punto (una línea de código) en la que le dices al depurador que pare para seguir paso a paso o comprobar el estado de las variables.

8. Documentación.

Al documentar, hay que añadir explicaciones a todo lo que no es evidente. No hay que repetir lo que se hace: Hay que explicar por qué se hace.

Y eso se traduce en:

- ¿De qué se encarga una clase? ¿Un paquete?
- ¿Qué hace un método?
- ¿Cuál es el uso esperado de un método?
- ¿Para qué se usa una variable?
- ¿Cuál es el uso esperado de una variable?
- ¿Qué algoritmo estamos usando? ¿De dónde lo hemos sacado?
- ¿Qué limitaciones tiene el algoritmo? ¿... la implementación?
- ¿Qué se debería mejorar ... si hubiera más tiempo?

La forma más habitual de documentar es a través del uso de comentarios en el código, pero...

¿Cuándo hay que documentar?

Por obligación:

1. al principio de cada clase
2. al principio de cada método
3. ante cada variable de clase

Por conveniencia :

1. al principio de fragmento de código no evidente
2. a lo largo de los bucles

Y por si acaso:

1. siempre que hagamos algo raro
2. siempre que el código no sea evidente

Es decir, que los comentarios más vale que sobren que que falten.

¿Cómo documentar el código?

En Java, tenemos muy fácilmente accesible la documentación con Javadoc. Antes de una clase o función/método, basta con escribir /** y pulsar enter, y generará las líneas básicas necesarias.

¿Es Javadoc la Única alternativa?

No. Existen varias, pero una de las más usadas, y muy parecida a Javadoc es Doxygen, con licencia GPL. doxygen.org

Ejemplo de documentación de clase con Javadoc.

```
/** Clase que representa a un objeto alumno.
 * @author Miguel Páramos
 * @version 1.0
 * @since 0.3
*/

Public class Alumno {
...
}
```

¿Qué directivas podemos usar?

- @author nombre

Se usa en declaraciones de interfaces y clases, para nombrar los autores, normalmente en orden cronológico de modificación, el primero el creador. Puede haber varias directivas @author en el mismo bloque de comentarios.

- @since versión

Se utiliza en declaraciones de interfaces y clases, indica la versión del código fuente donde fué introducida.

- @deprecated

Se utiliza en declaraciones de cualquier elemento, e indica que está obsoleto y no debe usarse, pero se mantiene por retrocompatibilidad.

- @param nombre descripción

Se utiliza en comentarios de funciones y métodos. Describe el uso del parámetro, e indica si tiene alguna característica especial.

- @return tipo descripción

Se utiliza en declaraciones de métodos y funciones, indica qué devuelve y qué características puede tener. El tipo no es obligatorio, pero ayuda.

- @throws excepción descripción

Indica excepciones que el método podría lanzar, y las razones para lanzarla.

- @see nombre

Tiene que poner una referencia a un método de esta u otra clase, con la nomenclatura Clase#método y genera un acceso directo en la documentación a la documentación de ese método.

Directivas específicas de Doxygen.

- @bug Descripción

Se utiliza en declaraciones de cualquier elemento, e indica que hay un bug a solucionar allí, se indica cada uno en una nueva línea.

- @brief descripción

En declaraciones de cualquier elemento, declara una pequeña descripción, se puede usar en lugar de poner el texto directamente, como en el ejemplo de hace unas transparencias.

- @pre Descripción

Se utiliza en funciones y métodos, e indica alguna precondición que se ha de cumplir para llamarlo. Cada una en una nueva etiqueta @pre

- @post Descripción

Se utiliza en cualquier elemento, pero normalmente en declaraciones de funciones y métodos. Indica alguna postcondición que cumplirá el método al salir, o su elemento @ret.

- @copyright Descripción

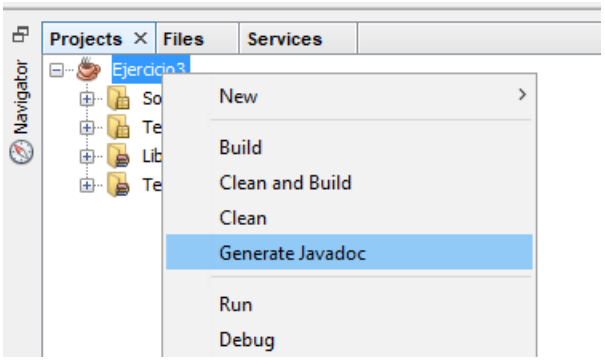
Se utiliza en cualquier elemento, pero normalmente en declaraciones de clases, ayuda a indicar la licencia del código.

- @todo Descripción

Se utiliza en cualquier elemento, indicando cosas que faltan por hacer aquí, se puede usar también en comentarios de una sola línea.

Cómo generar un Javadoc.

Una vez que has terminado de documentar, toda la documentación que has hecho te sirve para crear una página web específica para tu código, con el formato de documentación de java. Para esto, tendrás que hacer click derecho con el ratón en el proyecto, y seleccionar “Generar Javadoc”.



Al hacerlo, si no hay errores, se te creará la carpeta dist / javadoc, que contendrá la página web.

TEMA 4

Unidad 4: Estructuras de Almacenamiento

1. Operaciones con String.

String es un tipo que se comporta como un tipo de dato básico, pero en realidad no lo es: Puede asignarse a null. Lo veremos más adelante. Lo distinguimos de los tipos básicos porque no hay forma abreviada de escribirlo con minúscula, como pasa por ejemplo con int. Representa una secuencia de letras en un orden, desde la que está en la posición cero, hasta la que está en la posición length-1. Trabaja con tamaños muy grandes, por lo que normalmente no hay que preocuparse por hacer Strings demasiado grandes.

Tiene funciones disponibles tanto si usamos el nombre de clase como el nombre de fichero para llamarlas. **Podemos hacer:**
String.
o:
String miString="";
miString.
Y ver las funciones disponibles.

charAt

charAt, recibe un único parámetro, y devuelve el carácter que hay en esa posición específica. No nos podemos pasar de la longitud del array menos uno. Se empieza a contar desde cero.

Length

length, devuelve el número de letras que tiene un String, muy útil para comprobar tamaño o recorrer letra a letra con bucles. **Ejemplo:**

```
String texto="Batalla de paintball mañana";
//Devuelve 0 porque son iguales.
	System.out.println("CompareTo: "+texto.compareTo("Batalla de paintball mañana"));
 //Devuelve un negativo por la s de mañanas.
 System.out.println("CompareTo: "+texto.compareTo("Batalla de paintball mañanas"));
//Devuelve positivo por la n y la t que compara en “paintball”.
	System.out.println("CompareTo: "+texto.compareTo("Batalla de paintball mañanas!!!"));
//Devuelve negativo porque la c es más grande que el espacio.
	System.out.println("CompareTo: "+texto.compareTo("Batalla de paintball mañana"));
//Devuelve negativo porque B es más pequeño que b.
	System.out.println("CompareTo: "+texto.compareTo("batalla de paintball mañana"));
```

toLowerCase / toUpperCase

toLowerCase, convierte todas las letras de un string a minúscula. No lo modifica, devuelve otro string distinto modificado:

```
        String texto="Batalla de Paintball mañANa";
        System.out.println(texto.toLowerCase());
//Imprime “batalla de paintball mañana”.
```

Si quisiera modificar texto tendría que hacer:

```
texto=texto.toLowerCase();
```

Tenemos la función contraria, toUpperCase, que convierte todo a mayúsculas. No modifica la variable que llama a no ser que se asigne.

```
String texto="Batalla de Paintball mañANa";
        System.out.println(texto.toUpperCase());
//imprime “BATALLA DE PAINTBALL MAÑANA”.
```

Contains

contains, devuelve true si la cadena que llama al método contiene la cadena que se le pasa por parámetro, es decir si la de parámetro es una subcadena de la que llama. Tiene que estar completamente. **Ej:**

```
String texto="Batalla de Paintball mañANa";
        //Devuelve true: encuentra int en paintball.
        System.out.println(texto.contains("int"));
        //Devuelve true, porque esa cadena pertenece al texto.
        System.out.println(texto.contains(" de Paint"));
        //Devuelve false, porque una p es mayúscula y la otra minúscula.
        System.out.println(texto.contains(" de paint"));
        //Devuelve true porque son la misma cadena, y una cadena se contiene a sí misma
        System.out.println(texto.contains("Batalla de Paintball mañANa"));
```

indexOf

indexOf, devuelve el número de letra (empezando a contar por cero) donde se encuentra por primera vez la cadena que viene por parámetro en la cadena que llama a la función. Devuelve -1 si no la encuentra. **Ejemplo:**

```
String texto="Batalla de Paintball mañANa";
        //Devuelve 11.
        System.out.println(texto.indexOf("Paintball"));
        //Devuelve 10 porque incluye el espacio.
        System.out.println(texto.indexOf(" Paintball"));
        //Devuelve 16, que es donde empieza.
        System.out.println(texto.indexOf("ball mañ"));
        //Devuelve -1 si no la encuentra.
        System.out.println(texto.indexOf("patata"));
```

Tenemos también lastIndexOf, que devuelve la última ocurrencia, en lugar de la primera.

endsWith, devuelve un boolean que indica si el String que llama a la función acaba exactamente por el string que se le pasa por parámetro.

startsWith que te permite buscar a partir de un número de letra:

```
        String texto="Batalla de Paintball mañANa";
    System.out.println(texto.startsWith("ANa")); //False: La contiene pero no empieza por ahí.
    System.out.println(texto.startsWith("Bata")); //True
    System.out.println(texto.startsWith(texto)); //True, porque un string empieza por sí mismo.
    System.out.println(texto.startsWith("ANa")); //False: La contiene pero no empieza por ahí.
```

Equals

equals compara los textos, y devuelve true en caso de que sean exactamente iguales, false en caso contrario. Se debe usar siempre equals y nunca == o != con Strings.

equalsIgnoreCase es igual que equals, pero ignora la diferencia entre mayúsculas y minúsculas.

```
String texto="Batalla de Paintball mañANa "+"con caramelos";

    //True
    System.out.println(texto.equals("Batalla de Paintball mañANa con caramelos"));
    //False
    System.out.println(texto.equals("Batalla de paintball mañANa con caramelos"));
    //True
    System.out.println(texto.equalsIgnoreCase("Batalla de paintball mañANa con caramelos"));
```

Replace

replace recibe dos strings por parámetros: El primero es el texto que busca, el segundo, es el texto con el que sustituye lo que ha buscado en el String que llamó a la función. No modifica el String: Devuelve otro modificado. Ejemplos:

```
    String texto="Adêle la cigüeña es una inútil";
    //Se reemplaza inútil por enérgica.
    System.out.println(texto.replace("inútil","enérgica"));
    //Se reemplazan todas las letras a por una i.
    System.out.println(texto.replace("a","i"));
```

replaceAll funciona de una forma similar, si no usamos expresiones regulares. (Ya aprenderemos lo que son).

subString

substring recibe dos enteros: La posición inicial y la posición final, y del string que se ha usado para llamar a la función se devuelve como string todas las letras que haya entre esas dos posiciones sin incluir la última. Hay que asegurarse de no poner posiciones más grandes que el tamaño del string. Si solo se usa un argumento se coge el substring desde la posición que indica el argumento hasta el final del String:

```
String texto2="HolaQueHacesPorAqui";
    System.out.println(texto2.substring(7)); //Devuelve HacesPorAqui
    System.out.println(texto2.substring(7,12)); //Devuelve Haces
```

Trim y Split

trim no recibe parámetros, y elimina todos los espacios y tabulaciones al principio y al final de un string. Ejemplos:

```
System.out.println("Hola que tal?     ".trim()); //Imprime Hola que tal?
 System.out.println("       ¿Todo bien?   ".trim()); //Imprime ¿Todo bien?
```

split nos separa un string a partir del carácter que digamos, en varios String, devolviéndonos un String[] (Array de String), que veremos lo que es en el siguiente punto. Hay varios String almacenados en la misma variable. **Al hacer:**

“Hola, Buenos días”.split(“ “); obtenemos tres String, resultado de separar por espacios: “Hola,”,”Buenos” y “días”.

Por último, convertir de Char a String se hace:
O sumando la cadena vacía por delante o por detrás:

```
String sa=a+"";
```

Si queremos convertir de String a char, tenemos que usar charAt.

2. Arrays

2.1. Definición.

Un Array o Vector es una estructura que permite almacenar varios valores del mismo tipo bajo el mismo nombre de variable. Tiene además otra diferencia importante con las variables que conocemos hasta ahora: No es un tipo básico, sino un conjunto de ellos (o un conjunto de elementos de otro tipo, lo veremos en el próximo tema).

2.2. Declaración, acceso y modificación.

Necesitamos saber cuántos de estos elementos habrá en el conjunto como máximo. Lo tenemos que saber a la hora de inicializar la variable.

Las variables que son arrays se declaran poniendo los corchetes [] detrás del tipo que vamos a declarar. **Por ejemplo:**

```
// si un entero se declara:
int valor;
// Un array de enteros se declara:
int[] valor;
```

Con esto queremos decir que esta variable “valor” no va a contener un solo entero, sino un conjunto de ellos. **Ejemplo:** int[] valor = new int[12];

Al declararlo como hemos hecho, se rellena el array con variables con el valor por defecto del tipo de la variable. En este caso, al ser un array de enteros, se reservan dos ceros.

También existen los literales de Array, que se escriben entre { y }, y contienen literales (o nombres de variables, o expresiones) del tipo declarado en el array, separados por comas. **Ej:** String[] solo4alumnos = {"William","Sara","Dani","Grego" };

Acceder a cada posición del array se hace con un número entero entre los corchetes. Esto especifica que de todas las posiciones del array, se accederá a la primera. Si el array es de String, cada una de estas posiciones es un String. **Ej:** solo4alumnos[0];

Asignar un valor a una posición del array se hace tratando cada posición como lo que es, un valor del tipo del que se declara el Array. Si tenemos un char[] miCharArray, en cada posición se puede almacenar un valor de tipo char. **Ejemplo:** miCharArray[2] = ’c’;

Cuando declaramos una variable, le decíamos a la máquina virtual de java “Oye, acuérdate de que hay cuatro bytes para un entero que se va a llamar contador, y que está aquí en este hueco de RAM”.

Java sabe que un int siempre ocupa 4 Bytes, por lo que sabe el tamaño que va a necesitar para almacenar la variable. Eso mismo pasa con el resto de tipos que conocemos hasta ahora (menos String, pero java lo enmascara para que no se note).

Como hasta que no lo inicializamos no sabremos cuántos int voy a querer reservar, el sistema sólo se guarda una referencia (o puntero) a la dirección de memoria donde voy a empezar a reservar. En el momento de inicializar, es cuando se reservan tantos conjuntos de cuatro bytes como enteros yo vaya a reservar.

2.3. Recorriendo Arrays.

Recorrer un array se hace normalmente con un bucle for, porque siempre sabemos su tamaño. Se recorre desde cero hasta el tamaño menos uno
(o < lenght), para no pasarnos del tamaño. En este caso, es 5, así que tenemos posiciones de 0 a 4.

Cuando no inicializamos una posición del array, y la imprimimos, da el valor por defecto del tipo. En los tipos no básicos es null (no hay nada). Si a una variable que vale null, le llamamos a una función (toUpperCase en este caso) da un NullPointerException.

2.4. Paso de arrays a funciones.

Cuando paso un int a una función se comporta así:

```
main {
        int a = 3;
        modificaInt(a);
}
public static void
modificaInt(int b) {
        ...
}
```

Dentro de la función b:

```
public static void
modificaInt(int b){
        b = 5;
}
```

Cuando termina la función, b deja de existir, y no ha cambiado.
Sin embargo, cuando **paso un array a una función:**

```
int[] array = new int[3];
modificaArray(array);
public static void
modificaArray(int[] b){
        (Modifico b por dentro, por ejemplo, poniendo un 2 en la posición 1)
}
```

Cuando termine la función, se va a destruir la variable b, pero la flechita de array es la misma.
Se modificó el valor de la posición 1 del array b. Pero como b era una copia de array, y copiar punteros es copiar hacia dónde apuntan sus flechitas… las dos flechitas apuntaban a las mismas direcciones de memoria. Era lo mismo modificar la variable array que modificar la variable b.

2.5. Devolver arrays en funciones.

Para decirle a una función que devuelva un array, hay que hacer lo mismo que con cualquier otro tipo de dato: Poner el tipo justo antes del nombre de la función, en su cabecera. **Por ejemplo:**

```
public static int[] mitadOriginal(int[] orig){
```

2.6. Intercambio de valores.

Intercambio de valores:

```
char[] pacman = {'Ʊ','©','©','©','©','©','©'};
```

El intercambio de valores es mecánico: Siempre se hace igual

Paso 1. me guardo uno de los valores a intercambiar en una variable auxiliar.

```
char aux = pacman[0];
```

Paso 2. la posición de la variable que he guardado como auxiliar la asigno al valor de la posición con la que quiero intercambiar

```
pacman[0] = pacman[1];
```

Paso 3. la posición de la variable que no he modificado, la cambio por el valor auxiliar

```
pacman[1] = aux;
```

Puedo ver el resultado impreso:

```
for (int i = 0; i < pacman.length; i++) {
    System.out.print(pacman[i] + "   ");
}
```

2.7. Valores extremo.

Podemos rellenar todas las posiciones de un array con el mismo valor usando Arrays.fill. El primer argumento es un array (se modifica) y el segundo el elemento con el que queremos rellenarlo. La función no devuelve nada. Ej:

```
Arrays.fill(pacman,'©');
```

La alternativa sería un for:

```
for(int i = 0 ; i < pacman.length ; i++) {
        pacman[i] = '©';
}
```

Si intentamos ir atrás con ‘a’ en la posición cero, o adelante con ‘d’ en la posición length()-1, nos revienta con un indexOutOfBoundsException. Porque no existe ninguna posición anterior a la primera, ni después de la última con las que intercambiar. Este problema es muy común en los valores extremo del array (la posición cero, y la última posición, length-1)

Tenemos dos estrategias posibles para impedir la Excepción:

- No seguir avanzando si intento salirme
- “Circular” en el array: Si intento ir adelante en la última, volver a la primera. Si intento ir atrás en la primera, volver a la última.

Siempre que queramos modificar valores de un array, tenemos que tener cuidado de tener en cuenta estos valores extremos, con una estrategia u otra. Cuando obtenemos errores indexOutOfBoundsException, lo primero de lo debemos sospechar es que estamos haciendo algo mal en valores extremos.

2.8. Bucle for alternativo.

Por último, vamos a ver una forma alternativa de hacer un bucle for, que se puede usar con los arrays. Ese bucle se suele llamar bucle foreach, en lenguajes como php se pone foreach(...), y en otros como java se sigue usando la palabra for.

Se construye asi:

```
for( TipoElemento nombreElemento : array) {
        cuerpo del bucle, en el que en lugar de usar array[i] , uso
        directamente nombreElemento.
}
```

Ejemplo:	**Es lo mismo que:**

`for(String actual:array){`
`  System.out.println(actual);`
`}`
	`for (int i = 0 ; i < array.length ; i++) {`
	`  System.out.println(array[i]);`
	`}`

3. Argumentos de programa

Los argumentos de programa son datos separados por espacios que se pasan al programa a través de la línea de comandos. Pueden ser ninguno, uno, o los que queramos. Y su comportamiento no está definido de forma estándar: Nosotros decidimos en nuestro programa cuáles son y cómo se comportan. Es importante entonces documentarlos.
La única regla que se sigue es que normalmente antes de dar un argumento que represente un valor, se da un argumento que empieza por -, e indica el tipo de valor que se va a dar.
EJ: mysql -u miusuario -p “aaaa”
indica a la llamada a mysql que el usuario es miusuario y la contraseña “aaaa”, porque en el manual de mysql pone que el argumento detrás de -u tiene que ser el nombre de usuario, y el que vaya después de -p, la contraseña.

En nuestros programas, capturamos argumentos a través del array de String que se recibe de parámetro en la función main. Normalmente se llama args, pero no es obligatorio.

```
public static void main(String[] args){
```

En la variable args, si no hay argumentos, tendréis un array de tamaño cero. Si hay argumentos, tendréis un array con el mismo tamaño que número de argumentos, con uno en cada posición. Viene ya inicializado desde la llamada, es decir, en ningún caso tenemos que inicializarlo nosotros con args=new String[x].
Con este código podemos listar todos los argumentos que se pasan al programa:

```
public static void main(String[] args) {
      System.out.println("Nº Argumentos: "+args.length);
      for (int i = 0; i < args.length; i++) {
         System.out.println("Argumento "+i+" = "+args[i]);
      }
   }
```

Para llamar al programa usando argumentos, podemos generar el .jar, y llamar a mi programa desde la línea de comandos con:

```
java -jar miprograma.jar argumento1 argumento2
```

Los argumentos son los textos separados por espacios que ponemos detrás del nombre de nuestro ejecutable jar.
Podemos hacer que varias palabras sean un solo argumento, poniéndolas entre comillas:

```
java -jar miprograma.jar argumento1 “este es el argumento 2” argumento3
```

Tened en cuenta que los argumentos siempre son de tipo String, porque vienen en un array de String, si queréis usarlos como números, hay que transformarlos.

Formas de uso:
Tenemos dos estrategias para capturar argumentos:

- La más fácil y menos segura: Dar un significado fijo a cada posición de argumentos.
- La más elaborada y más segura, poner argumentos precedidos de - (que es lo que se hace siempre por convención), y al siguiente argumento ya le queda establecido el valor, de esta forma podemos poner los argumentos en cualquier orden.

En la Alternativa 1: asignando significados fijos a cada posición de argumentos, podemos usarlos muy fácilmente, pero restringimos el programa a que se tenga que ejecutar por narices con esos argumentos. **Ejemplo con la calculadora:**

```
switch(args[0].toLowerCase()) {
    case "suma":
        float res = Float.parseFloat(args[1]) + Float.parseFloat(args[2]);
        System.out.println(args[1] + " + " + args[2] + " = " + res);
        break;
}
```

Solo podemos usar este programa llamandolo así:
java -jar miprograma.jar operacion operando1 operando2. **Ej:**
java -jar miprograma.jar resta 3 4.5
Si ponemos argumentos de menos, o les cambiamos el orden (ej: poniendo 3 4.5 suma), no va a funcionar bien. Si le pongo más argumentos de la cuenta, ignora los que sobran.
En resumen: ¡No uses esto!

En la Alternativa 2: indicando en argumentos que comienzan por -, el valor que va a venir en el siguiente argumento. No es obligatorio que empiece por - , pero es recomendable porque todo el mundo lo hace así, y es más fácil de entender para todos.

El primer paso es, antes de empezar el resto del programa, buscar entre los argumentos, a ver si está -l o -language. En ese caso, cogeremos como idioma el valor que indique el argumento siguiente:

```
public static void main(String[] args) {
    // Comprobamos si los argumentos documentados están.
    char idioma = 's'; // s significa español, e significa inglés.
    for (int i = 0 ; i < args.length ; i++) {
        if(args[i].toLowerCase().equals("-l") || args[i].toLowerCase().equals("-language")) {
            if(args[i+1].toLowerCase().equals("english")){idioma='e';}
           if(args[i+1].toLowerCase().equals("español")){idioma='s';}
        }
    }
```

Con esto, si llamo al programa: java -jar miprograma.jar -l english, la variable idioma se pone al valor ‘e’, y ya sé en el resto del programa que los textos van en inglés.

Ahora, cada vez que imprimo un texto en español, tengo que tener cuidado de hacer un if, si he elegido idioma ‘e’, imprimo en inglés, si elijo ‘s’, en español. Cada vez que saco un texto por pantalla. Este es el ejemplo de uno de esos casos:

```
if(idioma=='e'){
        System.out.println(“Choose an option…”);
} else if (idioma=='s') {
        System.out.println(“Elige una opción…”);
}
```

4. Matrices

4.1. Definición.

Las Matrices , definidas en una sola frase, son vectores que contienen vectores. Si un vector es una estructura que permite almacenar varios valores del mismo tipo, una matriz es una estructura que permite almacenar estructuras que almacenan varios valores del mismo tipo.
Tampoco son un tipo de dato básico, y también necesitaremos saber de antemano cuántas estructuras y cuántos elementos queremos almacenar. Resumido: es un conjunto de Arrays.

Se pueden definir tres tipos de matriz, según la relación entre las filas y las columnas:

- Regulares:
  - Cuadradas si todas las filas tienen el mismo número de columnas, y el número de columnas es igual al de filas.
  - Rectangulares si todas las filas tienen el mismo número de columnas, pero el número de columnas es distinto al de filas.
- Irregulares si cada fila tiene un número distinto de columnas.

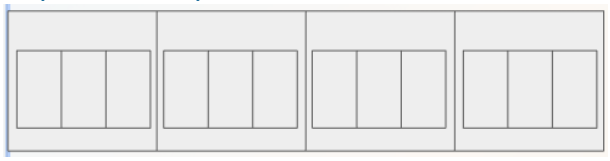
4.2. Declaración, modificación e impresión.

Si declaramos un vector de esta forma:
`int[] valor;`
Y una matriz es un vector que contiene vectores, e indicamos mediante los [] que algo puede contener varios valores, las matrices se declaran así:
`int[][] valor;`

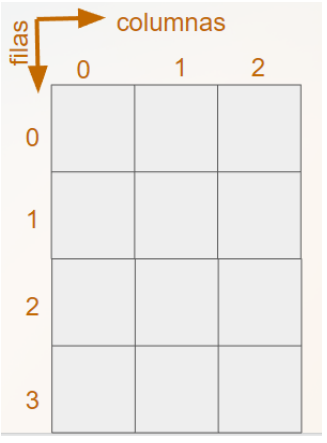
Una matriz entonces se inicializa de la siguiente forma:
`int[][] valor=new int[4][3];`
Esto indica que tendremos en la variable valor cuatro vectores (indicado entre los primeros corchetes), donde cada uno de ellos contiene tres enteros (indicado por los segundos corchetes). En total, tendremos 4*3=12 valores enteros en la matriz.

Si algo como esto: `int[] valor=new int[12];`
Lo representamos así:

Donde en cada cuadro cabe un entero, Esto: `int[][] valor=new int[4][3];`
Lo podemos representar así:



Donde en cada uno de los cuadros interiores, cabe un entero.
Aunque lo más típico es verlo de esta manera:
Como si fuera una tabla.
Hay dos formas de declarar matrices y darles valores.




```
int[][] matriz=new int[2][2];
 matriz[0][0]=1;
 matriz[0][1]=2;
 matriz[1][0]=3;
 matriz[1][1]=4;
```

Una matriz de 2x2 la podemos inicializar así:
`int[][] matriz={{1,2},{3,4}};`

Como los arrays que son, las matrices tienen un atributo length para averiguar su número de elementos. Como los arrays que son, cada una de las filas de una matriz tiene un atributo length para averiguar su número de elementos.

Podemos averiguar cuántas filas tiene una matriz si accedemos a:
matriz.length
Podemos averiguar cuántas columnas tiene una fila determinada (i) de una matriz, accediendo a:
matriz[i].length

Para imprimir los valores que contiene un array, necesitábamos un bucle for. Para imprimir los valores que contiene una matriz (un array que contiene arrays), necesitamos un bucle for dentro de otro bucle for.
¡Ojo a los errores comunes a la hora de recorrer todos los valores de una matriz! Sea para imprimirlos o para lo que los necesitemos. **Por ejemplo:**

```
int[][] matriz = {{1,2}, {3,4}, {5,6}};
for (int i = 0 ; i < matriz.length ; i++) {
    for (int j = 0 ; j < matriz[i].length ; j++) {
        System.out.print(matriz[i][j] + " ");
    }
    System.out.println("");
}
```

4.3. Paso de matrices a funciones.

Paso de matrices a funciones
Pasar una matriz a una función por argumentos me permite modificar una matriz desde dentro de la función, y los valores que modifique en ella dentro, permanecen modificados una vez salgo de la función. Ocurre por tanto lo mismo que con arrays.

En memoria, pasar una matriz como argumento a una función, funciona prácticamente igual que pasar un array.

4.4. Devolver matrices en funciones.

Para decirle a una función que devuelva una matriz, hay que hacer lo mismo que con cualquier otro tipo de dato: Poner el tipo justo antes del nombre de la función, en su cabecera. **Por ejemplo:**
public static String[][] cambioNombre(String[][] orig) { …

El proceso interno es bastante más complicado que esto, ya que son punteros que apuntan a otros punteros que apuntan ya a datos que son cambiados.

4.5. Intercambio de valores.

Intercambio de valores
Para intercambiar el valor de dos posiciones en una matriz, el procedimiento es exactamente el mismo que en un array, pero pensando en dos dimensiones:

1 - Guardo en una variable auxiliar uno de los valores que quiero intercambiar (a)

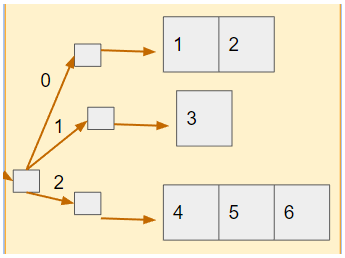
2 - Guardo en la posición que guardé antes en el auxiliar (a) el otro valor que quiero intercambiar (b)

3 - Guardo en el segundo valor (b) el valor que tenía en la variable auxiliar.

4.6. Matrices irregulares.

Las matrices irregulares no se diferencian en nada a las regulares aparte de la longitud de los vectores / arrays de su interior. **Por ejemplo:**
int[][] matriz = {{1,2},{3},{4,5,6}};

Se utilizan mucho en la creación de una lista de tareas.



4.7. Matrices multidimensionales.

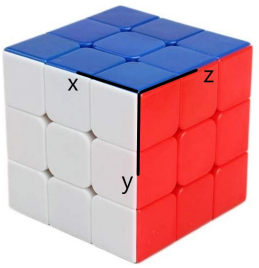
Matrices Multidimensionales
Hasta ahora hemos declarado arrays, y arrays de arrays (matrices). Pero no hay motivo por el que no podamos hacer arrays de arrays de arrays, o arrays de arrays de arrays de arrays. Empiezan a no tener nombre propio esas estructuras, pero nos pueden servir para cosas.

No existe límite entonces a cuántas anidaciones de array puedo hacer:
String[][][][][][][][] estructura;

Se inicializan así:
estructura = new String[2][3][2][1][3][2][1][2];

De las multidimensionales las más usadas son las tridimensionales, encima no son tan difíciles de imaginar.

matriz: [x][y][z];



4.8. Valores extremo.

En una matriz (matriz[3][4]) tenemos que tener cuidado con los valores extremos:
La máxima fila a la que podemos acceder es la fila nº (matriz.length-1).
Por lo de siempre: Empezamos a numerar las filas desde cero. **Ejemplo:** Si la longitud es 5, la máxima fila es 4. Si intentamos acceder a la fila 5, vamos a obtener un IndexOutOfBoundsException.
Con las columnas pasa igual, la máxima columna a la que podemos acceder es la columna matriz[fila].length-1. Y si nos pasamos de ese valor, obtendremos IndexOutOfBoundsException porque esa columna no está reservada en memoria.

Tenemos dos alternativas para manejar la situación, en que se pide ir más allá de alguno de los valores extremo:

1. Impedir que se acceda a esos valores.
2. Hacer un comportamiento circular: que lo que haya después del valor extremo al final sea el primer valor de la fila/columna, y que lo que haya antes del valor extremo 0 sea el último valor de la misma fila/columna.

En cualquiera de las dos alternativas, a los valores extremos hay que hacerles lo mismo: un if..else que contemple todos esos casos aparte, y se comporte el programa en consecuencia a lo que queramos.

5. Arrays, Matrices y Recursividad

Recursividad con Arrays.

Los arrays, las matrices y la recursividad nacieron para estar juntos.
Vamos a comenzar con los arrays. Sabemos recorrerlos con un **bucle for:**

```
char[] array = {'a','b','c','d','e','f'};
for (int i = 0; i < array.length; i++) {
    System.out.print(array[i]+"\t");
}
System.out.println("");
```

Y sabemos pasar eso a una función:

```
public static String imprimeArray(char[] arr){
      String res="";
      for (int i = 0 ; i < arr.length ; i++) {
         res += arr[i] + "\t";
      }
      res += "\n";
      return res;
   }
```

Para hacerlo recursivo teníamos una serie de pasos del tema anterior:

1. Pasamos a la función el contador y las variables que sean necesarias para la condición de parada.
2. El caso base es el contrario de la condición de parada.
3. En el caso contrario al base, el recursivo, se hace lo mismo que se estaba haciendo en el cuerpo del bucle.
4. Hacemos el incremento del contador, incrementándose para la llamada recursiva. Si estamos concatenando un String o acumulando un valor, lo concatenamos/acumulamos también aquí.
5. Para llamar a la función, desde el main, ponemos en el contador el mismo valor con el que lo inicializamos en el for.

Siguiendo estos pasos, imprimeArray nos quedaría de forma recursiva:

```
public static String imprimeArray(char[] arr, int cont){
      String res = "";
      if(cont >= arr.length) { // El contrario de la condición de parada.
         // Caso base. No hacemos nada en esta función en concreto.
      }else{ // Caso recursivo
         res += arr[cont] + "\t"; // Lo mismo que en el cuerpo del bucle.
         res = res + imprimeArray(arr,++cont); // Llamada recursiva incrementando contador.
      }
      return res;
   }
```

Y lo llamamos desde el main:

```
System.out.println(imprimeArray(array,0));
```

Recursividad con Matrices.

Siguiendo con las matrices, también se llevan muy bien con la recursividad, porque al fin y al cabo son arrays de arrays.

Podemos usar por ejemplo, la función imprimeArray recursiva que ya tenemos hecha, para recorrer cada fila de forma iterativa, e imprimir cada una de ellas de forma recursiva:

```
char[][] matriz = {{'a','b','c','d','e','f'},{'g','h','i','j','k','l'},{'m','n','o','p','q','r'}};
for (int i = 0 ; i < matriz.length ; i++) {
    System.out.println(imprimeArray(matriz[i],0));
}
```

En cada iteración, a imprimeArray le vamos poniendo el array que es cada una de las filas de la matriz: matriz[i].

De nuevo, podemos sacar esto a una función iterativa, para posteriormente hacerla recursiva, siguiendo los mismos pasos de siempre:

```
public static String imprimeMatriz(char[][] mat,int cont){
    String ret="";
    if(cont>=mat.length){
        //Caso base
    }else{
        ret += imprimeArray(mat[cont],0) + "\n";
        ret += imprimeMatriz(mat,cont+1);
    }

    return ret;
}
```

Con esto, con llamar a la orden siguiente, imprimimos:
`System.out.println(imprimeMatriz(matriz));`

El secreto de recorrer matrices recursivamente es primero pensar en el array que es cada una de las filas (y contiene las columnas), y hacer una función que te haga la tarea para un array. Después, pensar en la función recursiva que te trata cada uno de esos arrays como si fueran valores simples de un array (las filas).
Siguiendo los pasos de siempre, obtengo esta función recursiva:

```
public static byte vecesEnArray(String[] arr,byte cont,String nombreBuscado) {
    byte veces = 0;
    if(cont >= arr.length){
        //caso base
    } else {
        if(arr[cont].equalsIgnoreCase(nombreBuscado)) {
            veces++;
        }
        veces += vecesEnArray(arr,(byte)(cont+1),nombreBuscado);
}
```

Hay muchos otros casos para los que se utiliza recursividad con arrays y matrices: La búsqueda recursiva o la ordenación recursiva son muy eficientes y más fáciles de programar de forma recursiva. Hay muchos algoritmos para ambos usos.

Los tipos básicos una vez siendo arrays y ya no funcionan como tipos básicos.

int [] numero = new int [3];

[I@5acf9800
Hash del puntero (posición matemática). Imprime el puntero del array (valor de las casillas del array).

System.out.println(numero[0]);
Sysout de la posición 0 del array.

Los arrays no pueden contar desde los números negativos.

String[] nombre = {"Paco","Sergio","Ana","Carmen","Mario"};
short [] numeros = { 3,4,5,2,91,21,4212,24,1 }
boolean [] booleanos = { true , false , true , 3 > 39 , …}

12 / 01 / 23

Hay un for especial para los arrays, conocido generalmente como “foreach”.
En java se escribe:
for (float actual:array) {
ret += actual+”\t”;
}

La variable “actual” se va “actualizando” constantemente con cada vez que pasa por el bucle.

La desventaja es: el array no se puede modificar dentro de ese for.

20 / 01 / 23

Argumentos:

// Paso 1: Traerse la declaración de todas las variables que puedan coger su valor de argumentos al principio del programa, e inicializarlas a valores absurdos o imposibles.

// Paso 2: Al principio del programa, recorrer todos los argumentos, y si encontramos el argumento que necesitamos para cambiar el valor a una variable, se lo cambio.

// Paso 3: A la hora de pedirlo por teclado o darle un valor por defecto compruebo si sigue teniendo el valor absoluto que le puse. Si lo sigue teniendo, lo pido por teclado, si no es que ya me viene dado por los argumentos.

23 / 01 / 23

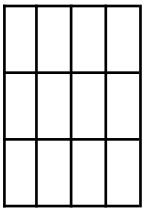
// Tipos de Datos numéricos:

```
byte num = 5;
byte[] array = {3,4,2};
byte[][] matriz = new byte[3][4];
byte [][][] matriz2 = new byte [3][3][3];
```

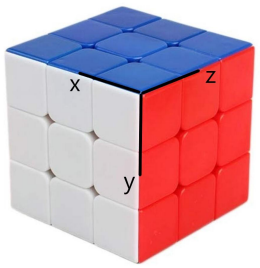
num:

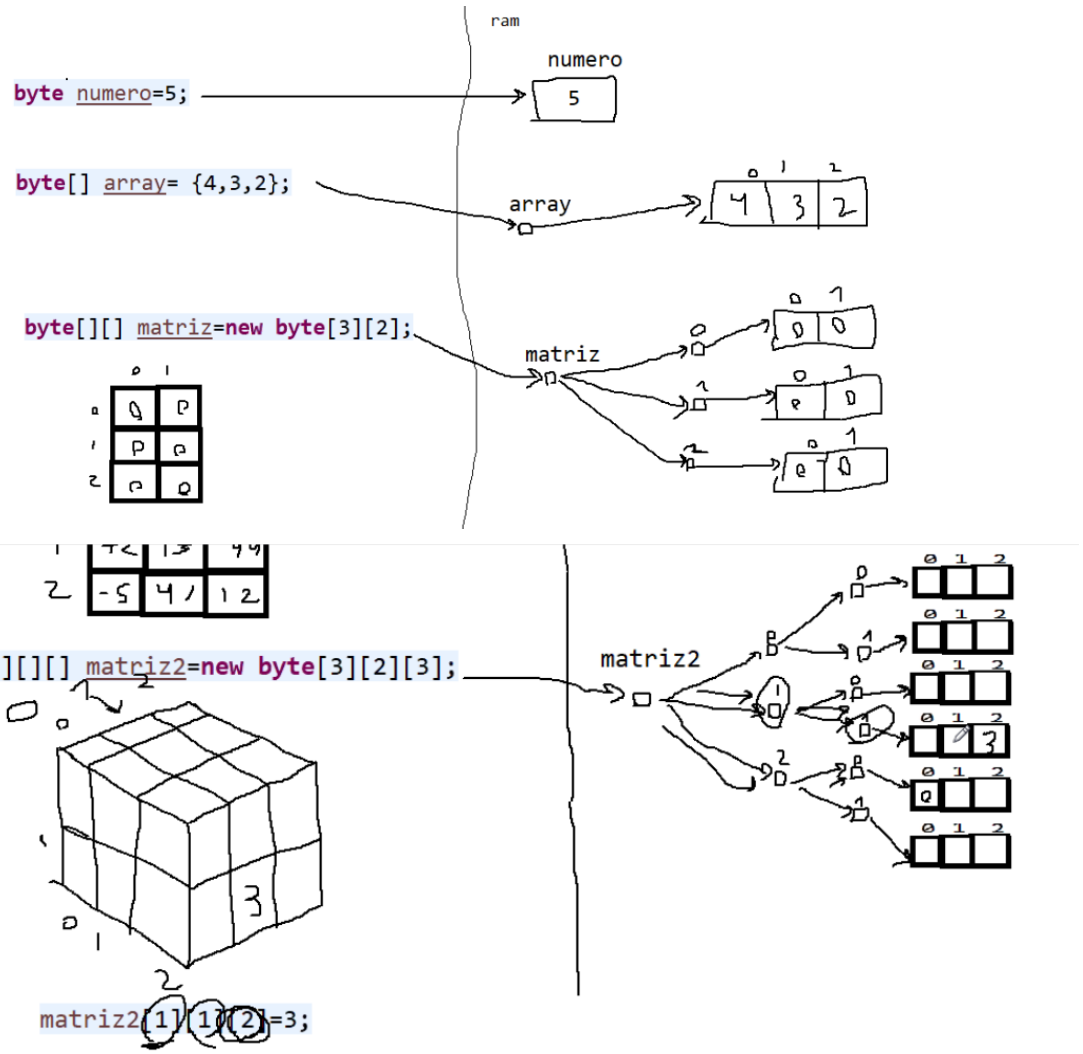
array:

matriz:



matriz2: [x][y][z]




ram
byte numero=5;
numero
5
byte[] array= {4,3,2};
array
byte[][] matriz=new byte[3][2];
matriz
byte[][][] matriz2=new byte[3][2][3];
matriz2
matriz2[1][1][2]=3;

```
// Tipos de Datos numéricos:

byte num = 5;
byte[] array = {3,4,2};
byte[][] matriz = new byte[3][4];
byte [][][] matriz2 = new byte [3][3][3];

 matriz [2][0] = 7;

for (byte x = 0 ; x < matriz.length ; x++) {
   for (byte y = 0 ; y < matriz[x].length ; y++) {
      matriz[x][y] = y;
      System.out.print(matriz[x][y] +  "\t");
   } // for
   System.out.println();
} // for

System.out.println("\n---------------------------------\n");

for (byte x = 0 ; x < matriz2.length ; x++) {
   System.out.println("Profundidad " + x + " -----");
      for (byte y = 0 ; y < matriz2[x].length ; y++) {
            for (byte z = 0 ; z < matriz2[x][y].length ; z++) {
               matriz2[x][y][z] = z;
               System.out.print(matriz2[x][y][z] + "\t");
            } // for
            System.out.println();
      } // for
  } // for
```