**Programación Multimedia y Dispositivos Móviles**

**Tema 0-Programación en Java**

1. **Sintaxis básica.**
2. **Variables.**
3. **Estructura if y operadores lógicos.**
4. **Estructura if - else y estructura if – else if**
5. **Bucles**
6. **Métodos**
7. **POO**
8. **Arrays**
9. **Herencia**
10. **Interfaces**
11. **Sintaxis básica**

Todo programa escrito en el lenguaje Java ha de incluir un “bloque” de código que se llama **clase**. Dicho bloque, viene determinado por la palabra reservada **class** y se marca su inicio y fin a partir del uso de llaves: { }

(foto 1)

Es decir, no podremos escribir código fuera de estas dos llaves, ha de estar incluido en este bloque.

A su vez, dentro del bloque **class**, en el desarrollo en Java es común incluir el **método** main, método desde el cual **comienza** a ejecutarse nuestro código.

(foto 2)

Para poder comprobar que, efectivamente, nuestro código empieza a ejecutarse desde dentro del bloque que representa el **método** main, vamos a introducir en él nuestra primera línea de código. Esta línea de código se va a encargar de lanzar un mensaje por **consola**.

Es imposible concebir la programación sin la existencia de una **consola**, entorno desde el cual podremos mostrar mensajes para comprobar el correcto funcionamiento de la aplicación, comprobar errores, etc. Es una herramienta con la cual ya debes de estar familiarizado.

En Java, para mostrar un mensaje por consola, utilizamos el método System.out.println, entre otros. Así, podremos comprobar el correcto funcionamiento de nuestro código con un mensaje simple:

(foto 3)

Por último, recordemos también la importancia que tiene incluir **comentarios** en nuestro código. De esta manera, podremos entenderlo mucho mejor cada vez que retomemos un trabajo que quedó a medias o que otros integrantes de nuestro equipo de desarrollo entiendan mejor nuestro código.

Existen los comentarios de línea, los cuales comienza con los caracteres //

(foto 4)

Al igual que los comentarios de párrafo, los cuales comienzan con los caracteres /\* y terminan con \*/

(foto 5)

1. **Variables**

A la hora de programar manejamos datos, los guardamos, cargamos y editamos. Para que esto sea posible, estos datos han de quedar almacenados en algún “lugar” de nuestro programa u ordenador.

Concretamente, estos datos quedan almacenados en la memoria RAM, reservando un espacio de memoria que en nuestro programa se representa como una **variable**.

En conclusión, siempre que necesitemos un dato nuevo en nuestro desarrollo, hemos de declarar una **variable** en la cual quede almacenado y representado dicho dato.

A la hora de declarar una nueva variable (un nuevo dato a utilizar), lo haremos a partir de la siguiente sintaxis:

**tipo** nombre;

* 1. Tipos de variables

Existen dos tipos diferenciados de variables, las variables de tipo **primitivo** y las variables de tipo **clase**.

Las variables de tipo primitivo son todas aquellas que representan un tipo de dato **simple**, tales como un número entero (int), un número decimal (float), un caracter (char), etc.

Se podría comparar con el modelado 3D, el cual parte con figuras **simples** para poder hacer el resto: cubos, esferas, cilindros, etc.

En nuestro IDE (Intelli J Idea) se ve claramente que tipos son primitivos porque estos vienen escritos en color naranja.

(foto 6)

Por otro lado, las variables tipo clase representarán datos más complejos, lo cual lo dejamos para más adelante. En el caso de nuestro IDE, diferenciaremos estos tipos porque vienen escritos en color blanco. Una variable tipo clase en Java, por ejemplo, es una variable tipo **String**.

(foto 7)

* 1. Declaración de variables

A la hora de declarar una variable, hay que tener ciertas cosas en cuenta:

* El nombre de la variable **debe** comenzar por una letra.
* El nombre **no debe** contener espacios.
* Se **aconseja** que se escriban en inglés, ya que es el lenguaje estándar en el mundo de la programación.
* Se **aconsejan** nombres prácticos/identificables.
* Se **aconseja** que comiencen por minúscula.
* Se **aconseja** no incluir caracteres extraños (aquí se incluyen caracteres propios de nuestro lenguaje, como las tildes o la “ñ”)
* Se **aconseja** que cada palabra (a excepción de la primera), comience en mayúscula. Por ejemplo: myLongVariableName. Esta escritura es conocida como *Lower Camel Case*, debido a su parecido, por las subidas y bajadas, con las jorobas de un camello.
  1. Operaciones básicas con variables

Las operaciones más simples que se pueden llevar a cabo con variables son las siguientes:

* Asignación ( = ) :

Esta es la operación más importante. Se suele confundir con el “igual” de matemáticas, pero no es lo mismo.

Por ejemplo, si tenemos una “variable1” que vale 10 y una “variable2” que vale 5, en matemáticas es una locura escribir: variable1 = variable2, ya que estamos diciendo que 10 es igual a 5.

En programación esto no es así, dicho símbolo se podría entender como “toma el valor de…”. Es decir, la variable descrita a la izquierda **toma el valor** **de** lo descrito a la derecha.

Siguiendo entonces el mismo ejemplo, nuestra “variable1” **toma** el valor de “variable2”, pasando a valer 5.

* Suma, resta, multiplicación, división y módulo ( + - \* / %):

Funcionan igual que en matemáticas, son operaciones simples entre dos valores numéricos. El resultado de dicha operación siempre ha de quedar **asignado** a una variable.

Es decir, no podemos realizar la sentencia: variable1 + variable2, ya que el resultado de esta suma **se pierde**, no queda almacenado en ningún sitio.

El resultado ha de quedar almacenado en otra variable, por ejemplo:

variableResultado = variable1 + variable2;

Cabe destacar el operador **módulo** (%). Este operador da como resultado el **resto** de una división.

* Concatenación ( + ): Hay ocasiones en las que el símbolo + significa otra cosa totalmente diferente. Si este símbolo es utilizado entre un valor **numérico** y un valor tipo **texto** (String), entonces estamos **concatenando** (pegando) ambas partes.

Por ejemplo, la instrucción: System.out.println(“Hello” + 5); nos saca por consola el mensaje: Hello5

* Incremento y decremento (++ --): Operaciones que se realizan sólo sobre variables de tipo entero. Esta operación lo que hace es incrementar o decrementar **en 1** el valor de una variable entera. Así, si variable1 = 5, al hacer variable1++, esta pasa a valer 6.
* Asignación compuesta (+= -= \*= /= %=): Hay muchas ocasiones en las que operamos una variable con otro valor con el objetivo de mantener el nuevo resultado en **la misma** variable. Por ejemplo, sumar 5 a variable1 y guardar el resultado obtenido en ella misma, variable1:

variable1= variable1 + 5;

Todo este tipo de operaciones, sea de suma, de resta, multiplicación o división, se puede “resumir” a partir de la **asignación compuesta**. Siguiendo el ejemplo anterior, quedaría:

variable1 += 5;

1. **Estructura if y operadores lógicos**
   1. Introducción a estructuras de control y comparadores

En un principio, nuestros “bloques” de código, tal y como sucede con el método main, se va a ejecutar de forma **secuencial**, es decir, interpretándose de arriba abajo. Primero se ejecuta la primera línea, después la siguiente, etc.

Podemos “cortar” esta secuencia a partir del uso de estructuras, estructuras que ya te sonarán tales como el **if, else, if – else if, while, for**, entre las más simples.

Todas estas estructuras se rigen por una o más **condiciones**, las cuales vienen descritas entre paréntesis ( ). Si la condición o condiciones se cumplen, entonces nuestro compilador, interpretará y ejecutará el **cuerpo** de la estructura, el cual viene marcado de principio a fin, de nuevo, con el uso de { }.

Y, ¿cómo formamos estas condiciones? A partir del uso de los siguientes **comparadores**:

* Mayor que: valor1 > valor2
* Menor que: valor1 < valor2
* Mayor o igual que: valor1 >= valor2
* Menor o igual que: valor1 <= valor2
* Igual que: valor1 == valor2
* Distinto de: valor1 != valor2

Un ejemplo de una estructura if, podría ser:

(foto 8)

* 1. Operadores lógicos

Los **operadores lógicos** nos sirven para poder **unir** condiciones. En este caso, sólo existen dos tipos, cuyo funcionamiento está basado en el Álgebra de Boole. Estos operadores se enumeran a continuación, junto con su notación:

* Operador AND: &&
* Operador OR: ||

Su funcionalidad se basa en poder concatenar o unir un conjunto de condiciones para poder establecer una única **condición resultante** o general.

Por ejemplo, se podría establecer que un robot sólo comprara café si su precio es menor a 2 euros **y si, además**, tiene más de 10 euros en el bolsillo. Como se puede comprobar, existen **dos condiciones**: La primera, que el precio del café no sobrepase los 2 euros. Por otro lado, que el robot tenga más de 10 euros en el bolsillo. En conclusión, para que el robot compre, se han de cumplir **ambas** condiciones, la una y la otra.

Este sería un claro ejemplo para utilizar el operador AND (&&), pudiendo establecer la condición del ejemplo como:

precio < 2 **&&** dineroEnBolsillo > 10

Por otro lado, el operador OR ( || ) funciona al contrario. Es decir, **basta con que una** de las “subcondiciones” se cumpla para que se cumpla la condición resultante. Es decir, la condición: precio < 2 || dineroEnBolsillo > 10, se cumpliría si el precio es menor a 2 **o** si el dinero que lleva el robot es superior a 10, una condición totalmente **independiente** de la otra.

Como se puede comprobar, las posibilidades que ofrecen los **comparadores** anteriormente vistos, con la unión de condiciones a partir de estos **operadores lógicos** pueden llegar a ser infinitas, pudiendo establecer cualquier tipo de condición lógica necesaria, por ejemplo:

precio < 2 && dineroEnBolsillo > 10 || tuDinero != miDinero

Este ejemplo puede ser enrevesado, pero se pueden encontrar patrones similares según vaya siendo necesario establecer condiciones más complejas si nuestro código de programación lo requiere. Ante la ilegibilidad de estas condiciones tan largas, lo recomendable sería **incluir paréntesis** para poder separar las “subcondiciones”, por ejemplo:

**(** precio < 2 && dineroEnBolsillo > 10 **)** || **(** tuDinero != miDinero **)**

Para resolver este tipo de condiciones, se llevará un orden basado en **prioridades**. En el siguiente listado quedan ordenanos los operadores lógicos desde la prioridad más alta a la más baja:

1. Operador de negación **!**
2. Operador AND **&&**
3. Operador OR **||**

El uso de los paréntesis, (), permite cambiar el orden impuesto por la prioridad de estos operadores.

1. **Estructura if-else y estructura if – else if**
   1. Estructura if – else

Como hemos visto, en ciertas ocasiones nos interesa ejecutar cierto código si se cumple una condición o condiciones (con un “if”) pero también nos puede interesar que se ejecute cierto código **si no** se cumplen.

En estos casos, utilizamos una estructura **if – else**, es decir, traducida como **si – si no**.

Esta estructura funciona por sí sola como un único bloque, es decir, no podemos incluir un **else** si justo antes no está incluida una estructura **if**.

Otra cosa a destacar es que el **else** no lleva consigo los paréntesis de condición, ( ). Esto es lógico, ya que un **else** (traducido como “si no…”) ya es una condición en sí mismo que sólo se cumple si el **if** previo no se ha cumplido.

Un claro ejemplo de esta estructura podría ser:

(foto 9)

* 1. Estructura if – else if

Hay ocasiones en las que se nos puede plantear un conjunto de estructuras **if,** una detrás de otra, por ejemplo, el algoritmo que se lleva a cabo para cambiar el color de la barra de vida de un personaje de un videojuego:

(foto 10)

El código se podría plantear de esta forma y funcionaría, ya que, en base a las vidas que lleva el personaje, se ejecuta un **if** u otro.

Lo que sí que podemos observar claramente es lo siguiente:

* Todos los if planteados forman parte de **un único** entendimiento u **algoritmo**. En este caso, manejan el color de una barra de vida.
* Si estamos en un rango de vida cualquiera (por ejemplo, de 20 a 50), **no tiene sentido** que el compilador revise el resto de posibilidades, no es lógico que estemos en otro rango de vida al mismo tiempo. Si dejamos que el compilador revise todos los if habiéndose cumplido ya uno, estamos malgastando recursos.
* Si las vidas bajan muy rápidamente desde otro script, es posible que este algoritmo acabe entrando en dos if al mismo tiempo, dando lugar a errores (estaríamos en dos rangos de vida al mismo tiempo)

Cuando estas circunstancias se cumplen, lo más sensato es utilizar una estructura **if – else if.**

La primera de las condiciones a comprobar se abre con un **if** y las siguientes se van enlazando con **else if** (es decir, algo así como: “**si no** se ha cumplido el anterior if, mira a ver **si** se cumple este”). Incluso, podemos llegar a cerrar todo el algoritmo con un **else**, cuyo cuerpo se ejecuta **si no** se ha cumplido ninguna condición de todas las anteriores.

Por último, también destacar que a la hora de utilizar cualquier estructura de las que estamos viendo, si en su cuerpo sólo existe **una línea**, podemos ahorrarnos las llaves, { }

(foto 11)

1. **Bucles**

Existen varios tipos de bucle utilizados en Java, siendo los más comunes el bucle while y el bucle for.

* 1. Bucle while

La sintaxis de un bucle **while** en Java es la siguiente:

(foto 12)

Como se puede comprobar, se escribe exactamente igual que un **if**, es decir, incluye paréntesis y llaves. En los paréntesis, al igual que sucedía con el **if**, se incluirá la **condición** o condiciones por las que se rige la estructura, mientras que entre las llaves irá **el código a ejecutar** si la condición o condiciones dentro de los paréntesis se cumple.

La diferencia está en que el cuerpo de esta estructura se podrá ejecutar **varias veces en bucle**. En un bucle while, su cuerpo se ejecutará una y otra vez **mientras** (while) la condición se cumpla.

Cuando el compilador accede a un bucle **while**, este se queda “encerrado” en el bucle **mientras** se siga cumpliendo la condición por la cual se rige. Veamos un ejemplo:

(foto 13)

En este ejemplo, inicialmente la condición del bucle se cumple, ya que la variable vidases mayor a 0. Es por ello, que, a partir de este bucle, mostraremos el mensaje “Estoy vivo” **infinitas** veces porque la condición se va a seguir cumpliendo.

Este comportamiento, por lo tanto, genera un **bucle infinito**, algo que tenemos que **evitar a toda costa**. Si en nuestro código tenemos un bucle infinito como es en este caso, el programa se **colgará** al quedarse el compilador encerrado en el bucle, ya que no puede ejecutar el resto del código.

Si el compilador ha accedido al interior del bucle es porque la condición se ha cumplido, entonces si la condición se sigue cumpliendo, ¿cómo “frenar” a ese bucle para que se deje de ejecutar de forma infinita? Simple, **modificando** el valor de la **variable** que **afecta** directamente a la **condición**.

Es decir, para este mismo ejemplo, si modificamos ya una vez dentro del bucle el valor de la variable vidas, es posible que consigamos hacer que el bucle deje de repetirse de forma infinita, por ejemplo, haciendo lo siguiente:

(foto 14)

Como se puede comprobar, por cada pasada o **iteración** del bucle, la variable “vidas” se decrementa en 10. Por lo tanto, llegará un momento en el que “vidas” deje de ser mayor que 0, haciendo que el bucle se **rompa** o termine y prosiguiendo con el código que venga a continuación. Este tipo de operación sobre la variable que rige el bucle, casi siempre suele efectuarse como **última instrucción** del cuerpo del bucle.

Teniendo ya este bucle funcional, si lo estudiamos a fondo, vemos como si lo ejecutamos, aparecería el mensaje “Estoy vivo” hasta 3 veces por consola. Los pasos que se realizan son los siguientes:

* “vidas” si es mayor a 0🡪 Se muestra el primer “Estoy vivo”. Variable “vidas” se decrementa en 10 y ahora vale 20.
* “vidas” si es mayor a 0🡪 Se muestra el segundo “Estoy vivo”. Variable “vidas” se decrementa en 10 y ahora vale 10.
* “vidas” si es mayor a 0🡪 Se muestra el tercer “Estoy vivo”. Variable “vidas” se decrementa en 10 y ahora vale 0.
* “vidas” **no** es mayor a 0🡪 Se romple el bucle y se seguirá ejecutando el código posterior.
  1. Bucle for

Al igual que el bucle **while** por su traducción es tratado como un bucle **mientras**, el bucle **for**, lo podríamos entender como bucle **durante** o bucle **para**. En español, la palabra “durante” tiene el mismo significado que “mientras”, y esto mismo sucede entre el **for** y el **while**: que son lo mismo. Es más, un bucle **while** puede ser transformado en un bucle **for** y viceversa.

Aun así, el bucle **for** tiene una diferencia frente al **while**. Utilizaremos un bucle **for** siempre y **cuando sepamos de antemano** cuantas veces queremos ejecutar el bucle. Si sucede lo contrario, sería más conveniente utilizar un while.

Otra de las diferencias principales de este bucle frente al while, es que este utilizará una variable entera para contar las **iteraciones** o pasadas del bucle. A esta variable se la denominará **variable de control**, comúnmente nombrada como “i”, de *iterator* o iterador.

Vamos a estudiar a fondo el funcionamiento de un bucle **for**. Para ello, nos apoyaremos en el siguiente ejemplo:

(foto 15)

Vemos que el bucle for **siempre** está constituido por tres partes bien diferenciadas y separadas por “;”, las cuales comentamos a continuación, en el mismo orden en el que aparecen en la imagen:

* **Inicialización de variable de control**: En esta primera parte es donde se **declara** la variable **de control** y en donde se le asigna un valor con el cual inicializar el bucle. Para este ejemplo, vemos como se declara la variable llamada “i” y que se inicializa a 0.
* **Condición de control:** Es la condición que va a controlar el bucle, al igual que la condición que utilizábamos en un while. Mientras se cumpla dicha condición, el bucle continúa. En este caso, vemos como la condición es que “i” sea menor a 10.
* **Paso:** Operación que se realiza sobre la variable de control y que va marcando el **paso** del bucle en cada iteración. En este caso, vemos como “i” se va incrementando en uno por cada **iteración** del bucle.

Ahora que ya hemos diferenciado las partes en las que se constituye un bucle for, vamos a estudiar su funcionamiento marcando los siguientes pasos:

1. Lo primero de todo, debemos fijarnos en la primera **iteración** como se inicializa la variable de control. En la mayoría de los casos se inicializa a 0, pero esto no tiene por qué ser así siempre.
2. **Al comienzo** de cada iteración, se comprueba si se cumple la **condición de control**. Si se cumple, pasamos a la siguiente iteración, si no, se rompe el bucle.
3. Se ejecuta el cuerpo de instrucciones del bucle, es decir, lo que lleve entre las llaves.
4. **Al final** de cada iteración se realiza la operación de **paso** sobre la variable de control.
5. **Se repiten** de forma continuada los pasos del 2 al 4, hasta que la condición del paso 2. deje de cumplirse.

Otra forma muy buena de interpretar un bucle for, es traduciéndolo como bucle “para”. Esta interpretación nace de su semejanza con una **sucesión matemática**.

Vamos a seguir el siguiente ejemplo:

(foto 16)

Se podría traducir como:

“Haz el interior del bucle **para** todos los valores de i posibles que van **desde 0** (inicialización) hasta **5** (condición) de **uno en uno** (paso)”

Hacer el interior del bucle **para** esos valores de i, es hacer la secuencia: 0, 1, 2, 3, 4, la cual se muestra por consola.

* 1. Bucles for anidados

Ahora que conocemos las estructuras de bucle, se nos abren infinitas posibilidades: un if dentro de un while, un for dentro del if, etc y también podría darse el caso de **un bucle dentro de un bucle**, por ejemplo, un for dentro de otro for.

Es importante saber cómo interpretar, ante todo, bucles for anidados, y para ello, nos apoyaremos en un ejemplo:

(foto 17)

Para este caso, ¿te atreves a decir cuántas veces se mostrará el mensaje por consola?

Si nos fijamos en el **bucle padre** (el que itera con la “i”) vemos como este se ejecuta 5 veces (desde i = 0 a i = 4, sin incluir 5). Esto hará que el bucle “hijo” (el que itera con la “j”) se ejecute esas 5 veces las veces que el mismo ya se repite de por sí. A su vez, el **bucle hijo** del bucle que itera con la “j” (es decir, el que itera con la “k”), se ejecuta 3 veces (para k = 0 hasta k = 2, sin incluir el 3).

En conclusión, este ejemplo se podría entender como “repite 5 veces lo que se repite 2 veces que a su vez repite un mensaje 3 veces”. Es decir, el mensaje se mostrará un total de 30 veces: 5 x 2 x 3.

1. **Métodos**

Los métodos son secciones de código que realizan **funcionalidades** muy concretas, es decir **hacen** una tarea concreta. Como hemos comentado anteriormente, en Java, toda clase empieza a ejecutarse por el método “main”. Esto no significa que sea el único método posible: podemos incluir otros ya predefinidos o generarnos uno propio.

Pensemos en las **acciones** que hace un personaje simple de un videojuego, por ejemplo, **moverse**, **disparar** y **tomar daño**. Estas acciones, pueden ser interpretadas en tres métodos diferenciados:

(foto 18)

Es muy importante respetar la sintaxis a la hora de escribir un método.

Por ahora, todos comenzarán por la palabra reservada **void**, seguido de su nombre (en Java es adecuado que comience en minúscula) y finalizado con unos paréntesis (en un principio, vacíos). Tampoco olvidemos el cuerpo del método, el cual es descrito entre las llaves, { }

El problema, es que, en Java, vemos como el método main trae consigo una palabra reservada **static**. El significado de esta palabra lo veremos más adelante. Debido a esto, también hemos de incluir en nuestros métodos dicha palabra reservada siempre y cuando queramos que sean ejecutados desde main()

(foto 19)

Como se puede comprobar, en nuestro IDE, los métodos nada más incluirlos, queda su nombre en gris. Esto es debido a que **no están siendo ejecutados**. Esto es muy importante: Un método **no** se ejecuta si no es llamado. Es decir, sólo lo hemos implementado, pero no estamos haciendo que se ejecute.

Para que se ejecute, basta con escribir en alguna parte del código, su nombre junto con los paréntesis. Esto es, siguiendo el ejemplo de la imagen:

moverse();

6.1. Parámetros de entrada

Cuando implementamos un método se nos da la libertad de poder replicar el algoritmo que lleva consigo pero para diferentes **valores posibles**. Es decir, siguiendo el ejemplo del método tomarDanho(), podemos comprobar como su algoritmo siempre es el mismo: Se restan a nuestras vidas **tantos puntos de daño recibidos** y se comprueba si las vidas han quedado por debajo de cero, en cuyo caso, estamos muertos.

Este algoritmo es idéntico para cuales quieran que sean los puntos de daño a recibir. Cuando esto nos ocurra, podemos **parametrizar** el método (darle parámetros) para que funcione en base a estas variaciones.

Para hacerlo, basta con incluir por la **entrada del método** (en los paréntesis), aquellos datos sujetos a variaciones en nuestro algoritmo. Estos datos han de ser especificados con un tipo y un nombre. Para nuestro ejemplo, incluimos por parámetro de entrada un dato entero **int puntosDanho**

(foto 20)

Ahora el algoritmo queda preparado para que pueda funcionar en base a cualquier valor para **puntosDanho**, pero, ¿dónde especificamos dicho valor? Lo especificaremos a la hora de llamar al método.

(foto 21)

Esto abre un abanico de infinitas posibilidades, ya que tenemos la libertad de ejecutar este método tantas veces como deseemos con infinitos valores diferentes de daño. De hecho, podemos ir incluso más allá, ya que se nos permite incluir **infinitos valores de entrada**, tantos como queramos.

Por ejemplo, podríamos hacer este algoritmo algo más complejo para que funcione también en base al número de enemigos que hacen daño, incluyendo así, hasta dos parámetros de entrada (separados por coma dentro del paréntesis):

(foto 22)

6.2. Parámetro de salida

Una forma muy buena de interpretar un método es viéndolo como si fuera una máquina de snacks. Para que la máquina de snacks funcione, esta precisa de **dos parámetros de entrada** (dinero y número de snack). Después, la máquina trabaja con esos datos (al igual que nuestro método tomarDanho) y, por último, **nos devuelve a cambio** nuestra bolsa de snacks. Esta bolsa sería el parámetro de salida del método.

Es decir, el parámetro de salida representa el resultado de haber ejecutado el método. Por ejemplo, en un método llamado sumarDosNumeros(), el resultado de haberlo ejecutado debería de ser el resultado de la suma. Otro ejemplo sería con nuestro método tomarDanho(). En este caso, el parámetro de salida podría ser la vida restante que nos queda tras haber sufrido el daño.

En todos estos casos, ese dato ha de ser **devuelto** (al igual que ocurre con la bolsa de snacks) pero, ¿cómo? Simplemente hemos de incluir la palabra reservada **return** seguida del dato a devolver. Es importante destacar que esta instrucción **siempre** irá al final del método.

(foto 23)

Después de haber implementado esta línea algo no ha ido bien, algo nos falta. Concretamente, el problema lo tenemos en la parte superior del método, donde figura su nombre. A la **izquierda** del nombre del método hemos de escribir el **tipo de dato** que se está devolviendo, en este caso, **int.**

(foto 24)

¿Por qué hacer este último paso es tan importante que si no lo hacemos el código no compila? Porque estamos constituyendo **la firma del método**, la cual se puede ver en la imagen anterior.

La firma del método es algo muy importante, ya que en ella va recogida **toda la información** necesaria para poder manejarnos con cualquier método que consultemos:

* Se nos especifica si es static o no, entre otros.
* Se nos especifica su nombre.
* Se nos especifica cuantos parámetros de entrada tiene y de qué tipo.
* Se nos especifica el tipo de dato de salida que devuelve.

La firma del método, es, básicamente, como unas **instrucciones** del mismo. A lo largo del curso consultaremos la firma de centenares de métodos, por lo que su comprensión es de vital importancia.

1. **Programación Orientada a Objetos (POO)**
   1. **Objetos definidos por propiedades y funcionalidades.**

¿Qué significa la POO? Y más concretamente, ¿Qué es para la POO un **objeto?**

En la POO existirán **objetos** o **instancias.** Estos objetos, por un lado, se definirán a partir de una serie de **atributos** o **propiedades** y, por otro lado, realizarán ciertas **funcionalidades**.

Con este modelo de abstracción, podemos ilustrar un ejemplo cotidiano:

* Objeto: piezaDeLegoRoja.
  + Propiedades o atributos: color, numAgujeros, ancho, alto.
  + Funcionalidades: montar, desmontar, rotar.
  1. **Clases**

Si seguimos “creando” objetos tales como piezaDeLegoAzul, piezaDeLegoAmarilla, miPiezaDeLego, tuPiezaDeLego, etc. nos damos cuenta de que todos estos objetos **comparten** las mismas **propiedades** y las mismas **funcionalidades**. Es decir, todos estos objetos se pueden definir por su color, su número de agujeros, su ancho y su alto, y, por otro lado, todas se montan, se desmontan y se rotan.

Quizás, otro objeto que “creemos” tal como un coche, **no comparta** estas propiedades y funcionalidades frente a las piezas de Lego. Es decir, un coche, aunque tiene color, ancho y alto, **no posee** un “número de agujeros” y las funcionalidades “montar”, “desmontar” y “rotar” aunque podamos pensar que un coche las posee, son procesos totalmente **diferentes** comparándolos con una pieza de Lego.

Es debido a estas peculiaridades que los objetos se clasificarán por grupos o **clases**. Es decir, los objetos piezaDeLegoRoja y piezaDeLegoAzul al definirse por las mismas **propiedades** y al tener las mismas **funcionalidades**, se puede decir que ambos pertenecen a la misma **clase**, que podríamos nombrarla para este ejemplo como la clase PiezasLego.

Si vamos más allá y entendemos las **clases** como fábricas, podríamos decir que sin clases no se generan objetos. Es decir, para que el objeto piezaDeLegoRoja exista, necesita **definirse** su clase (la fábrica para que dicha pieza se fabrique).

Estas fábricas (o clases) nos permitirán **instanciar** **infinitos** **objetos**, siempre y cuando estos objetos pertenezcan a la clase en sí, es decir, la clase PiezasLego no podrá generar **instancias** tales como un coche.

* 1. **Entendiendo scripts como clases.**

Vamos a fijarnos en un script recién creado y al que llamaremos PiezasLego:

(foto 25)

Vemos que, en el interior del script, justo antes del nombre aparece la palabra **class**, identificando así el script como la clase PiezasLego. También, creemos un nuevo script desde el cual poder generar tantas piezas de lego como queramos. Lo llamaremos CreadorPiezas.

Habíamos comentado que desde una clase podemos sacar **instancias** u **objetos infinitos**. Por lo tanto, si ya tenemos la clase, ¿cómo generamos estos objetos? Simple, tal y como hacíamos con las variables: Tipo + nombre. En este caso, **el tipo es el nombre de la clase en sí**. Por lo tanto, para generar el objeto piezaDeLegoRoja, haríamos lo siguiente en el script CreadorPiezas:

(foto 26)

Ahora, el objeto piezaDeLegoRoja está **declarado**,pero no tiene asignado ningún valor. ¿Cómo poder darle un valor a algo tan abstracto? No podemos darle un número como valor, por ejemplo. Recordemos que un objeto se define por sus **propiedades** y por las **funcionalidades** que realiza. Por lo tanto, ¿cómo podemos **editar** las propiedades de este objeto (su color, su alto, etc) y cómo podemos hacer que realice sus funcionalidades cuando nos interese?

Primero, vamos a definir esas propiedades y esas funcionalidades en el script apropiado, PiezasLego. Las propiedades, por un lado, serán las **variables** presentes en el script, mientras que las funcionalidades, serán los métodos.

Al definir, por lo tanto, las propiedades y funcionalidades que definen a los objetos de la clase PiezasLego, nos queda el script como sigue:

(foto 27)

Como se puede comprobar, podemos entender la clase PiezasLego como una **plantilla** que define a los objetos que van a salir de esa clase: se ve, por un lado, las propiedades que van a definir a los objetos creados y, por otro lado, las funcionalidades que pueden hacer.

* 1. **Creando instancias de la clase con constructores.**

Una vez creada la “plantilla”, ya está la fábrica preparada, ya podemos crear **objetos** de esta clase. De hecho, ya tenemos el primer objeto declarado, piezaDeLegoRoja, pero este **no tiene valor**. Es decir, como no se le ha **asignado** ningún valor (con “=”), queda con valor nulo (**null**).

**Todo** **objeto** en Java se le ha de dar un valor para poder trabajar con él. Esto lo haremos a través de un **constructor**, cuya **sintaxis** se muestra en el ejemplo siguiente en el que se crea el objeto llamado “miPieza”:

(foto 28)

Esta línea de código se ha de implementar **antes** de utilizar el objeto y en cualquier clase, ya sea otra diferente como por ejemplo clase *CreadorPiezas* o en su misma clase, *PiezasLego*.

El uso del constructor que acabamos de ver sólo “prepara” el molde para que el objeto pueda ser usado. Reserva cierto espacio en memoria para su uso, pero aún no se ha definido, ya que, por ejemplo, no hemos dado valor a sus **propiedades**.

* 1. **Manejo de propiedades y métodos de objetos.**

Una vez que hemos reservado espacio en memoria para nuestro objeto con el uso del **constructor**, para editar las **propiedades** de un objeto, basta con escribir un punto (“.”) justo después del nombre del objeto en cuestión y buscar la propiedad. De esta manera, podemos dar valor a cada una de las propiedades del objeto:

(foto 29)

Lo mismo sucede con las funcionalidades o **métodos**. Para hacer que un objeto realice una de sus **funcionalidades** basta con escribir el punto después del nombre del objeto y escribir dicho método:

(foto 30)

* 1. **Otros constructores**

Hemos visto como poder dar valor a un objeto a través del uso del **constructor**, pero este es sólo el constructor **por defecto**. Podemos crear más constructores, tantos como queramos y siempre los crearemos en la **clase** que los vaya a utilizar, en este caso, *PiezasLego*.

La sintaxis para crear un constructor es la siguiente:

*NivelProteccion NombreClase (param. entrada)*

*{*

*//Cuerpo del constructor*

*}*

Como se puede comprobar es similar a la estructura de un **método**, pero es importante diferenciarlos. Un buen truco para diferenciarlos, es fijarnos en la parte izquierda de su declaración. Los métodos **siempre** tienen un **tipo de dato** en su salida (ya sea “void” u otro), pero los **constructores** no, siempre comienzan con el **nombre de la clase** (obviando el nivel de protección, que la gran mayoría de las veces es *public*).

El propósito de hacer más constructores es para que, a la par que se reserva espacio en memoria para el objeto, se **seteen** (se les de valor) **sus propiedades**. Las propiedades a “setear”, vendrán por **parámetro de entrada** y se les dará valor en el **cuerpo** del constructor.

Se podrán hacer tantos constructores como **combinaciones** permitan sus propiedades y **no** pueden existir dos constructores idénticos.

Siguiendo el ejemplo de nuestra clase *PiezasLego*, esto son algunos de los constructores que se pueden declarar en base a las propiedades que posee esta clase. Uno de ellos incluye todas las propiedades, otro sólo el ancho y el alto y, por último, otro sólo el color:

(foto 31)

Sus respectivos usos a la hora de **crear** objetos utilizando estos constructores, serían los siguientes:

(foto 32)

Es importante recalcar que, si se genera al menos **un constructor** como hemos hecho, ya **no se puede** utilizar el constructor por defecto.

1. **Arrays**

Los **arrays** o matrices son otro tipo de variable más, como pueden ser el **int** o el **bool**. En matemáticas, una matriz es un conjunto de elementos que quedan organizados por filas y columnas.

En la programación un array es básicamente lo mismo, pero nos vamos a centrar en los arrays **unidimensionales**, es decir, serán un conjunto de elementos que quedan organizados en **fila**.

Es muy importante destacar que estos elementos **siempre** han de ser del mismo tipo, es decir, no podemos definir un array que contenga elementos tipo **int** con elementos tipo **bool.**

Por ejemplo, y a modo ilustrativo, la siguiente imagen representaría un array de 9 enteros:

(foto 33).

Vemos como se disponen en fila los distintos elementos del array, siendo cada uno de ellos una cuadrícula. Es importante destacar también, como cada uno de ellos almacena un valor entero diferente y que, además, cada uno viene representado por un **índice**, teniendo **siempre** como primer índice el índice 0.

Al ser el primer índice un 0 y ser un array de 9 elementos, entonces el último elemento adquiere el índice 8. Esto, aunque suene evidente suele ser uno de los **principales problemas** cuando trabajamos con arrays, ya que nos hemos acostumbrado desde siempre a contar desde 1.

No nos olvidemos que los arrays son un tipo más de variables, y que, por lo tanto, necesitan **declararse** en el código. Para declararlos, se utiliza cierta **sintaxis**, que se puede estudiar a partir de este ejemplo:

(foto 34).

Como se puede comprobar, la **sintaxis** es **muy similar** a cuando declarábamos un objeto. Tiene un tipo (en este caso, int[]), un nombre, y, después, hemos de darle un valor por defecto a través de un **constructor**. Con este constructor estamos definiendo **la longitud** (el nº de elementos) del array. En este caso, es un array de longitud 9.

Si utilizamos esta sintaxis, cada componente del array (cada “cajita”) recibe un **valor por defecto**. En el caso de un array de enteros, como es este caso, cada cajita tendrá como valor un **0**. En el caso de un array de Strings, cada cajita tendrá valor por defecto un string **vacío**, “” y así con el resto de tipos.

Otra forma de declarar un array es la siguiente:

(foto 35)

Para este caso, vemos que la declaración **a la izquierda** del igual (“=”) se mantiene **igual** que la anterior que hemos visto, lo que cambia es lo que viene posterior al “=”. En este caso, vemos que se asigna directamente un valor para cada componente, definiendo a su vez, la **longitud** del array. Nótese para este caso el uso de las llaves, ‘{ }’, en vez del uso de corchetes, ‘[ ]’, y como los elementos son separados con **comas**.

Desde un comienzo, hemos hablado de que los elementos de un array vienen numerados por índices. La razón detrás de esto es porque desde el código podemos **acceder** a los diferentes **elementos** que conforman un array para poder **operar** con ellos. Por lo tanto, para **hacer referencia** a uno de los elementos de un array, basta con escribir el nombre del mismo seguido del índice, tal y como se ilustra en estos ejemplos:

(foto 36).

Esto nos abre casi infinitas posibilidades a la hora de operar con los elementos de un array, pero hay que dejar en claro que sólo podemos operar **a nivel de elementos**, nunca a nivel de arrays (por ejemplo, sumar un array entero con otro array, de golpe).

También tendremos que tener cuidado a la hora de acceder a estos elementos. Si intentamos acceder a un elemento que no existe (por ejemplo, el índice 5 en un array de longitud 3), en un principio no ocurrirá nada, pero al ejecutar el código, tendremos un error grave y el programa se **parará por completo**.

Puedes ver ejemplos de cómo operar con los elementos de un array a lo largo de los **vídeos** de este tema.

1. **Herencia**

Ahora que entendemos en qué consiste la **programación orientada a objetos** y sabemos manejarla, podemos estudiar conceptos **más complejos** que giran en torno a esta forma de programar.

La **herencia** en **POO** se basa en “emparentar” **clases** entre sí, de tal manera que los **objetos** de estas puedan tener “rasgos” en común. Al fin y al cabo, es similar a una **herencia** de un ser vivo. Por ejemplo, una persona tiene el pelo rizado porque lo ha **heredado** de su madre. Una persona se comporta de cierta manera o hace ciertas **funciones** porque las **hereda** de sus progenitores.

Por lo tanto, en programación, cuando una ClaseA **hereda** de una ClaseB, los **objetos** de la ClaseA poseen **las mismas propiedades y mismos métodos** que la ClaseB, pero **no** a la inversa. Para hacer que una clase **herede** de otra, se utiliza la palabra reservada *extends*, como en el siguiente ejemplo:

(foto 37)

Por ejemplo, existe una clase *Enemigo* con un método *atacar()*. También tendremos las clases *Zombie* y *Dragón* las cuales **heredan** de *Enemigo*. Aunque en *Zombie* o *Dragon* **no** se defina ningún método, los **objetos** de estas clases podrán utilizar el método *atacar()* sin ningún problema, porque lo **heredan**.

Aunque esta jerarquía esté bien planteada en un principio, sí que es verdad que la forma de atacar de un zombie frente a un dragón es **diferente**. Es por ello que en las clases **hijas** (Zombie y Dragon) podemos **sobreescribir** al completo los métodos heredados para manejarlos de otra forma, o, utilizarlos al completo, pero **añadiendo** ciertas funcionalidades extras. Estas combinaciones se pueden realizar gracias a la palabra reservada *super*, de la cual puedes ver más ejemplos en los **vídeos** de este tema.

(foto 38)

Por otro lado, las propiedades **heredadas** tendrán los valores definidos en la clase **padre**, aunque, por supuesto, también se pueden sobreescribir con nuevos valores.

1. **Interfaces**

La forma en la que proceder a la hora de manejar interfaces en Java, es bastante similar al tema de la herencia, vista en el anterior apartado.

Una **interfaz** define un conjunto de **métodos** que ha de utilizar **obligatoriamente** cualquier objeto de una clase que la **implementa**. De esta forma, estaremos **obligando** a que los objetos de cierta clase en concreto, o, incluso de varias clases, utilicen el mismo conjunto de métodos.

Para poder hacer uso de una interfaz, haremos uso de la palabra reservada *implements*. De esta forma, en el siguiente ejemplo, se detalla como una clase **Zombie** implementa la interfaz llamada **Acciones**

(foto 39)

De esta manera, lo que estamos consiguiendo es que, cada vez que generemos un objeto nuevo de la clase **Zombie**, este, obligatoriamente ha de ejecutar obligatoriamente los métodos que se definan en la misma.

La forma de definir ese conjunto de métodos en la interfaz es la siguiente:

(foto 40)

Hay que destacar varias cosas. La primera de ellas, es ver como esto ya no es una clase con la palabra reservada **class**, si no **interface**. Por otro lado, también se puede comprobar que los métodos **no** tienen cuerpo (es decir, { }), sino que simplemente incluyen su nombre y sus **parámetros** (tanto de entrada como salida).

Al hacer esto, estaremos **obligando** a la clase Zombie ha que **complete** los métodos descritos en la interfaz:

(foto 41)

Vemos como, en este caso, efectivamente los métodos tienen **cuerpo** y podrán implementarse como deseemos.

Como conclusión, el uso de interfaces nos sirve para **organizar** mejor nuestro código, tener claro qué han de cumplir ciertos objetos y ofrecen un conjunto de ventajas. Por ejemplo, una clase puede implementar **varias** interfaces, cosa que no podemos hacer con el tema de la herencia (sólo existe una clase padre). También, una interfaz puede ser **compartida** por varias clases. Por último, si lo necesitáramos, una clase puede a la par **heredar** de otra e **implementar** interfaces, tantas como se necesiten.

En el siguiente ejemplo, se detalla como nuestra clase Zombie, mantiene la herencia descrita en el anterior apartado a la par que implementa la interfaz:

(foto 42)

Es importante destacar que, en estos casos, primero incluiremos la herencia con **extends** y, después, la implementación de la interfaz con **implements**.