

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Tema 4 – Introducción a la programación orientada a objetos

Introducción

En la programación clásica (lenguajes procedimentales) se tiende a modularizar las funciones en librerías que cada vez crecen más y más llegando un momento que los programas se convierten en entes realmente complejos de depurar, mejorar o cambiar.

En situaciones así conviene usar la programación orientada a objetos (POO) que utiliza recursos de abstracción para tener que manejar la menor cantidad de elementos posibles de un elemento complejo.

Esta filosofía está tomada de la interrelación de objetos del mundo real. Por ejemplo en un coche, como conductores del mismo, manejamos un número limitado de interfaces: volante, pedales, cambio, intermitentes,... y con eso somos capaces de manejar toda la complejidad interna del objeto coche.

En el mundo real los objetos son entidades que tienen una serie de propiedades que los describen y que responden a una serie de estímulos y realizan una serie de acciones. No importa cómo realiza esas acciones por dentro, sino solo la interfaz para manejar el objeto.

De esta forma podemos tener un objeto definido de la siguiente forma:

- Propiedades que lo describen.
- Objetos de los que a su vez se compone.
- Acciones que puede realizar.
- Estímulos a los que responde.

Estos elementos los iremos traduciendo al mundo de la programación para crear aplicaciones compuestas de objetos que se interrelacionen entre ellos.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Algunos conceptos de la POO

Clase

Es la definición genérica de un tipo de objeto. No se puede usar directamente porque es sólo una definición.

Ejemplos:

La definición del coche Citroën C4 consta de una serie de manuales que describen todos los elementos de dicho coche. Dicha definición está en la fábrica de Citroën y es única para todos los C4 que existen en el mundo.

La definición de una casa son los planos y toda la documentación de calidades y elementos de construcción de la misma.

Es por tanto un contenedor donde colocamos los distintos elementos (miembros) que van a conformar el objeto. Nos encontraremos:

- **Propiedades o atributos:** En principio serán variables que describen alguna cualidad del objeto. En el caso del coche puede ser el color, tamaño, peso, velocidad en un momento dado, consumo, etc... En algunos lenguajes de programación se permite que al mismo tiempo que se le da un valor a una propiedad o que se lee el valor de la propiedad se ejecute cierto código. En Java esto se puede simular mediante métodos.
- **Otros objetos:** Un objeto, además de tener propiedades que lo describen está compuesto por otros objetos. Esto se denomina **agregación**. Se consideran también propiedades del objeto.
- **Métodos:** Acciones que puede realizar un objeto. Serán funciones internas que realizan algún tipo de procesado con sus propiedades y, si fuera necesario, con parámetros externos.
- **Eventos:** Sucesos o estímulos que provocan una respuesta en un objeto. Es decir, que se ejecute una o varias funciones.

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Programación					CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR		Francisco Bellas Aláez (Curro)					

Diagrama de Clases UML

Asociado al diseño OO se puede usar la esquemática UML para representar las clases y las relaciones entre ellas. Iremos viendo a lo largo del tema distintos elementos usados en esta esquemática.

Como primer esquema tenemos el recuadro para representar una clase tal cual se ve a continuación:

Nombre de la clase
+Atributos
+Métodos()

Tiene tres partes donde se coloca respectivamente el nombre, las propiedades y los métodos de dicha clase.

Por ejemplo:

Coche
+color: String +numeroDePlazas: int +potencia: double +cilindrada: double +matricula: String +...
+Acelerar(): void +Frenar(): void +obtenerVelocidad(): double +llenarDeposito(litros:double): void +...()

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Objeto

Es la realización de una clase (también se denomina **instancia de una clase**). Si decíamos que la clase era una definición, el objeto es lo que se construye a partir de esa definición.

Ejemplo: El Citroën C4 matrícula 0666 WTF es un objeto creado a partir de la clase que es su definición. De hecho cada uno de los coches que vemos son los objetos instanciados a partir de su clase que es la definición.

Otro ejemplo es el caso de una casa. La clase sería el plano de la casa que diseña el arquitecto, y el objeto sería la casa ya fabricada. De hecho se pueden hacer varias casas a partir del mismo plano.

Por tanto, como regla general (que tendrá excepciones) cuando definimos una clase no se puede usar hasta que la **instanciamos** (creamos el objeto). En programación la instanciación equivaldrá a que el objeto se crea a partir de la plantilla "clase" y ocupa cierta memoria. Es decir, hasta que se reserva memoria para él y pueda existir.

Evento

Acción que recae sobre un objeto. No la realiza el objeto si no que la recibe. El objeto normalmente responde con un método a ese evento. En esto entraremos en 3ª evaluación, con la programación orientada a eventos.

Ejemplos:

En un coche: Apretar el acelerador, el elemento externo es el conductor que provoca el evento → Como respuesta el objeto coche responde con el método Acelerar (procesa el evento)

En un botón (por ejemplo de Guardar): Clic del ratón → Como respuesta el botón tendrá asociado un método que provoque la grabación de los datos en el disco duro.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Encapsulado

En POO se encapsulan el código que maneja los datos. Cuando construimos un objeto habrá partes del mismo (propiedades y métodos) que sean elementos de interfaz con otras partes del programa y otros elementos que sean puramente internos del objeto y solo se pueden usar dentro del mismo.

Ejemplo: Es equivalente al coche, en el que se encapsula como está construido por dentro y además no importa. Un manguito, una correa... Funciona (casi siempre...).

Otros ejemplos de encapsulamiento: El ser humano o un ordenador (Capas: HW, BIOS, SO, Aplicaciones) o los niveles de red OSI.

Evidentemente existe formas limitadas de acceder al funcionamiento (volante, palanca de cambios, etc.)

Ocultación de la complejidad: Definición de variables, objetos y métodos como:

Públicos: Se puede acceder desde cualquier parte. Ojo estas deben ser las justas para el manejo del objeto. Conforman su interfaz con el mundo.

Privados: Solo se puede acceder a ellas dentro del objeto.

Dependiendo del lenguaje de programación existen otras variantes a las dos anteriores.

Ejemplo: En un coche, la propiedad color o posición del volante serían "públicas". Sin embargo, la propiedad de tensión de la correa de distribución, flujo de gasolina o desplazar válvula serían privados (nosotros, si no somos mecánicos, no tenemos acceso a modificar esas propiedades o llamar a esos métodos).

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Clases y objetos en Java (y UML)

A continuación comenzaremos a aplicar los conceptos anteriores mediante lenguaje Java e iremos viendo como se pueden representar mediante la denominada esquemática UML (Unified Modeling Language). Lo primero que haremos es definir una clase para crear luego objetos a partir de la misma.

Recuerda que una clase vienen a ser los planos, la plantilla o las instrucciones a partir de la cual luego se puede crear el objeto.

Para definir una **clase** se utiliza la palabra reservada *class* y se le da un **nombre empezando por mayúscula**. Los distintos miembros de la clase (elementos que la componen) se sitúan entre llaves:

```
class NombreClase
{
    miembrosDeLaClase
}
```

Los miembros serán en un principio las propiedades y los métodos (variables y funciones). **Sus nombres empiezan por minúscula**.

La forma de acceder a los miembros de la clase es a través del operador punto:

```
NombreClase.miembro
nombreObjeto.miembro
```

Ejemplo de esto que ya hemos usado son los siguientes casos:

```
Math.PI //Math es el nombre de una clase, PI es una propiedad. No hay objeto porque PI es estático, esto ya lo veremos.
```

```
sc.nextLine(); // sc es un objeto que se instancia a partir de la clase Scanner. nextLine() es un método de dicho objeto.
```

Al principio puede parecer extraño el hecho de acceder al miembro de una clase cuando indicamos que es la plantilla para crear objetos, pero en ocasiones esto puede tener su utilidad pues hay cosas definidas desde la definición de la clase.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Por ejemplo el caso de la clase Citroen C4. En este caso se podría declarar como estático el miembro Marca (Citroen) porque todos los objetos tienen la misma marca (y si la comprara otra empresa cambiaría para todos).

Hagamos el siguiente ejemplo para la creación de una clase:

- Crea en una carpeta nueva denominada POO y dentro una clase con el mismo nombre y que contenga un main.
- En otro archivo en la misma carpeta, crea la clase Perro con el siguiente código.

Quedarían así:

// Archivo Perro.java

```
public class Perro {
    public String raza;
    public String nombre;
    private int edad;

    public int getEdad() {
        return edad;
    }

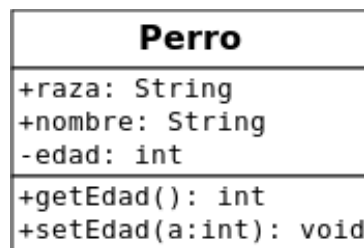
    public void setEdad(int a) {
        edad = a;
    }
}
```

// Archivo POO.java

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

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

El diagrama de Clases del caso anterior sería (Obviando la clase POO donde está el main):



- Es importante fijarse que **la clase que contiene el main es pública**. Esto es imprescindible, ya que es el SO el que accede al main y por tanto debe ser público y estar en clase pública.

Se puede añadir otra clase en el mismo archivo, pero **solo puede haber una clase pública por archivo**. Si se quiere más de una pública deben estar en distintos archivos.

Por eso lo habitual es poner **una clase por archivo**.


- Añade el siguiente código dentro del *main*.

```
Perro perro;
perro.
```

- En el momento que añadimos un punto, aparecen las distintas posibilidades de lo que podemos escribir a continuación. Es notable fijarse que aparecen nombre y raza pero no edad. ¿Por qué? Sin embargo también aparecen los métodos.
- Esto también funciona con otras clases ya usadas. Pruébalo con Math o con System.out.
- Completa el código (o casi):

```
Perro perro;

perro.raza="Can Palleiro";
perro.nombre="Terminator";
```


	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```
perro.setEdad(5);
System.out.printf("Tengo un %s llamado %s de %d años\n",
    perro.raza, perro.nombre,
    perro.getEdad());
```

Haciendo esto aparecen errores de compilación en cada "perro". ¿A que se debe esto?

En principio el error es que la variable perro no está asignada. Esto se puede "solucionar" asignándole el valor null.

```
perro = null;
```

Ahora ya no da error de compilación, pero si lo ejecutas saltará un error tipo **NullPointerException**. Veremos qué significa todo esto.

Una variable que "guarda" el contenido de un objeto como es en este caso perro, realmente esa variable solo es capaz de almacenar una dirección de memoria, es lo que se llama una referencia a memoria o un puntero a memoria. En POO se suele usar la terminología referencia a memoria o referencia a un objeto.

La palabra reservada **null** sirve para indicar una dirección de memoria no inicializada. Es una especie de "cero" para indicar que un objeto no referencia ninguna zona de memoria.

Entonces si sólo almacena una dirección ¿Dónde está toda la información del objeto? Pues a partir de la dirección guardada en la variable referencia. De esta forma, cuando declaramos una referencia a un objeto, lo único que hacemos es guardar los suficientes bytes (serían 4 bytes en una arquitectura de 32 bits y 8 en 64 bits) para almacenar la dirección donde va a ir el objeto, pero no el objeto en sí.

Por tanto perro es la **referencia al objeto**, pero el objeto en sí mismo está en otra zona de memoria.

Para reservar memoria para el objeto es necesario usar el comando *new* seguido por una función cuyo nombre es igual que el de la clase y se denomina **constructor**. Es decir, tendríamos algo así:

```
Perro perro;
perro = new Perro();
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

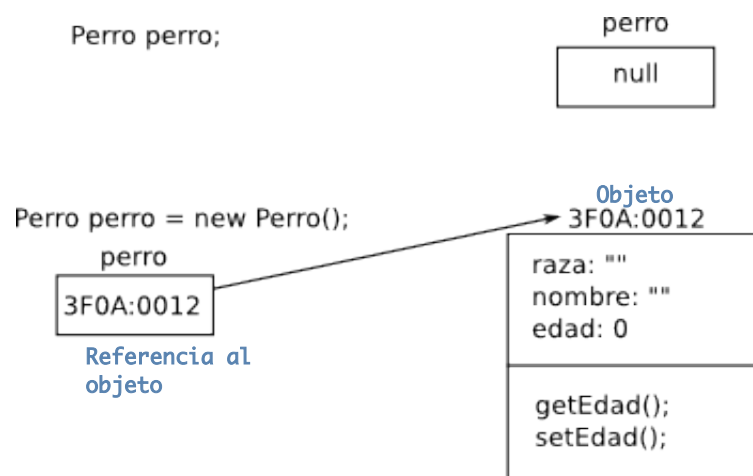
o escrito todo en la misma línea (**declaración + instanciación**):

```
Perro perro = new Perro();
```

Esto es lo que se denomina **instanciación** o creación de un objeto a partir de una clase. Es lo que ya hacíamos para usar un Scanner.

Ejecuta ahora y confirma que se ha solucionado el problema.


El siguiente esquema trata de clarificar lo explicado anteriormente:



A perro se le llama **variable de referencia al objeto** o directamente referencia y a la zona de memoria **donde están los datos se le denomina objeto**.

Como curiosidad en una aplicación Java los objetos se suelen guardar en una región de memoria denominada **Heap o Montículo**. Existe otra zona denominada **Stack** que es donde se suelen guardar las variables locales, las referencias, los valores de retorno de una función y parámetros.

La clase Perro podría colocarse en un archivo aparte dentro del mismo directorio que la principal o en un *package* y usar un *import* en la principal. Lo veremos más adelante.

	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Programación					CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR		Francisco Bellas Aláez (Curro)					

A continuación se explican algunos elementos de la orientación a objetos que encontramos en el programa anterior:

Encapsulado: Modificadores de acceso

Se puede observar que en los métodos y propiedades anteriores se antepone las palabras **public** o **private**. En el primer caso indica que dicho miembro es accesible desde el código escrito dentro o fuera de la clase incluyendo referencias que haya en otros paquetes (**packages**).

En el caso de ser **private** sólo es accesible desde dentro de la clase, nadie más ve el miembro.

Si no se especifica nada (**default**), se puede acceder al elemento desde cualquier clase del mismo paquete (directorio en el que está).

Veremos en el futuro un modificador más.

En el caso de **UML**, los miembros públicos se representan anteponiendo un + delante y los privados con un - .

En la visibilidad por defecto (visibilidad de paquete) se antepone ~.

Mark	Visibility type
+	Public
#	Protected
-	Private
~	Package

Marcas UML para la visibilidad. En un futuro entenderemos el significado de **protected**.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Métodos set y get (funciones de acceso)

La mayor parte de las veces cuando se realiza diseño orientado a objetos, las variables que son las propiedades del objeto se declaran privadas de forma que nadie las pueda modificar u obtener directamente si no a partir de unas funciones denominadas **set y get (setters y getters o accessors)**.

La función **set** se usa para **establecer** un valor de una propiedad, y la función **get** para **obtener** el valor de la propiedad.

Esto da al diseñador del objeto un mayor control sobre las propiedades de forma que cuando sea necesario leerla o escribir en ella se pueda realizar cierto código añadido para efectuar alguna acción paralela, alguna comprobación o algún otro cambio interesante a la hora de trabajar con dicha propiedad.

En el ejemplo del perro está reflejado en **setEdad()** y **getEdad()**. Las funciones en este caso no hacen nada más pero podrían aumentarse en un futuro para darle otras posibilidades. Por ejemplo podríamos establecer el setEdad() de la siguiente forma:

```
public void setEdad(int a) {
    if (a > 0) {
        edad = a;
    } else {
        edad = 0;
    }
}
```

De esta forma se introduce una comprobación que se hará siempre que se necesite establecer la edad del objeto.

Otra funcionalidad de los set es establecer de forma automática el valor de ciertas variables. Por ejemplo: supón que en la clase perro tenemos la propiedad temperatura. Y sucede que cada vez que cambia la temperatura, si es necesario, activamos o desactivamos una booleana que indica si hay o no fiebre.

Podría ser algo así:

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```
private double temperatura;

public void setTemperatura(double a) {
    // Se guarda la temperatura
    temperatura = a;

    // Y se cambia el valor de fiebre.
    if (a > 39){
        fiebre = true;
    } else {
        fiebre = false;
    }
    // Se podría poner, y sería mejor: temperatura = a > 39;
}
```

En este caso, en **fiebre** tendríamos **solo get**, pues el set sobra al cambiar automáticamente con temperatura.

En el caso de que estemos encapsulando una variable booleana con set/get, el estándar indica que el get debe denominarse **is**, por ejemplo:

```
private boolean fiebre;

public boolean isFiebre() {
    return fiebre;
}
```

La palabra reservada **this**

En ocasiones es necesario referirse dentro del código de la clase al objeto instanciado a partir de la misma. Sería como el pronombre personal **Yo** o alguna variante como **mi** para que un objeto se refiera a sí mismo. Para ello se utiliza la palabra reservada **this**.

Una de las utilidades es distinguir el nombre del parámetro y el nombre de la propiedad de la clase. Por ejemplo, podríamos cambiar la función *setEdad()* de la siguiente forma:

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```
public void setEdad(int edad) {
    if (edad > 0) {
        this.edad = edad;
    } else {
        this.edad = 0;
    }
}
```

Si no tuviéramos *this* no podría tener el mismo nombre el parámetro de la función y el campo de la clase.

Veremos distintas aplicaciones de *this* a lo largo del ciclo, aunque quizá la más evidente es poder pasar como parámetro al propio objeto dentro del código del propio objeto. O incluso devolver el propio objeto haciendo un **return this;**

Por ejemplo imagina que existe una clase Dueño y que tiene un método denominado Acariciar a la que hay que indicarle un perro. La cabecera de la función podría ser:

```
public void acariciar(Perro p) {... //En Dueño
```

Y cuando dentro de alguna función de la clase Perro se desee que el propio perro sea acariciado por su dueño podría hacerse así:

```
dueño.acariciar(this); //En Perro
```

Veremos más sobre objetos como parámetros más adelante.

Puedes pensar en *this* como en una especie de pronombre. Cuando alguien pronuncia *Yo* o *Mío*, significa algo distinto a que si lo pronuncia otra persona. Con *this* ocurre algo similar, es el *Yo* del objeto.

Constructores

Como se acaba de ver, para instanciar un objeto, se utiliza el operador `new` seguido de un método cuyo nombre coincide con el de la clase. Este método es el denominado **Constructor**. También hemos utilizado esto cuando creábamos el objeto `sc` de la clase `Scanner`.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Un constructor es un método cuyo **nombre coincide con el de la clase** y que inicializa los distintos miembros del nuevo objeto si fuera necesario. Se le pueden pasar parámetros.

El operador new reserva memoria para un objeto, llama al constructor y devuelve una referencia a la dirección de memoria dónde se ha creado el objeto.

El constructor **no requiere tipo de dato devuelto** (ni siquiera void), pues se llama a través de new, que va a devolver la referencia al objeto.

Si no se especifica un constructor, Java crea uno por defecto sin ningún parámetro. Pero en el momento que se crea uno, ya no existe el constructor por defecto. Veamos esto con un ejemplo. En la clase anterior añade este constructor:

```
public Perro(int edad, String raza, String nombre) {
    setEdad(edad);
    this.raza=raza;
    this.nombre=nombre;
}
```

Fíjate en el uso de *this* y en el hecho de que no tiene tipo devuelto.

Al hacer esto habrá un error en el programa principal pues se llama a un constructor sin parámetros. Para solucionarlo hay que llamar al nuevo constructor o añadir un constructor sin parámetros. En el primer caso el *main* quedaría así:

```
Perro perro = new Perro(5, "Can Palleiro", "Terminator");
```

```
System.out.printf("Tengo un %s llamado %s de %d años",
    perro.raza, perro.nombre,
    perro.muestraEdad());
```

En el segundo no tocaríamos el *main* pero a la clase Perro se le añadiría, por ejemplo, el siguiente constructor:

```
public Perro() {
    setEdad(0);
    this.raza="";
    this.nombre="";
}
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Por supuesto en lugar de usar setEdad podríamos poner directamente la propiedad:

```
this.edad=0;
```

Existe desde **Java 14** una nueva funcionalidad que es la denominada **Records** que es una forma sencilla de crear una clase simple para guardar datos de forma que se genera automáticamente el constructor y los setters y getters. No la usaremos pero puedes leer más sobre esto en:

<https://gerardo.dev/java14-records.html>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Sobrecarga

Existe una característica habitual en los lenguajes OO que es la posibilidad de llamar a varias funciones con el mismo nombre pero distinta lista de parámetros. Esto se denomina sobrecarga del método.

Acabamos de ver un ejemplo con dos constructores, uno sin parámetros y otro con 3 parámetros. Podríamos pensar que es un caso especial para los constructores, pero no es así.

Un ejemplo de uso es la función `println`, a la cual se le pueden pasar distintos tipos de datos con los que funciona (observa en tu IDE el autocompletado de `println`).

Veamos otro ejemplo creado por nosotros añadiendo tres nuevos métodos a la clase `Perro`:

```
public void ladrar() {
    System.out.println("GUAAU!!!");
}
public void ladrar(int n) {
    for (int i = 0; i < n; i++) {
        ladrar();
    }
}
public void ladrar(char idioma) {
    switch (idioma) {
        case 'I', 'i': // Inglés
            System.out.println("BARK!!!");
            break;
        case 'F', 'f': // Francés
            System.out.println("WOF!!!");
            break;
        case 'A', 'a': // Alemán
            System.out.println("KETEMUERDEN!!!");
            break;
        default:
            ladrar();
    }
}
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Al escribir en el main

```
perro.ladRAR(
```

aparecen las tres opciones. Esta es una forma de simplificar y tener menos nombres para hacer varias cosas.

La condición para que existan varios métodos es que haya distinta cantidad de parámetros o que los parámetros sean de distintos tipos. El valor devuelto por la función no influye en la sobrecarga.



COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Miembros estáticos

Hasta ahora hemos visto que para poder utilizar una clase hay que crear el objeto como instancia de esa clase con el operador *new* y luego acceder a sus miembros. Sin embargo, si haces memoria esto no siempre lo has hecho así. Por ejemplo has usado la función *sqrt()* de la clase *Math* sin instanciar ningún objeto de la misma, o la constante *PI* de la misma clase. Esto se puede hacer usando lo que se denominan miembros estáticos o de clase (en lugar de miembros de instancia u objeto).

Un miembro estático es común para todos los objetos de una misma clase y se puede acceder a él a través del nombre de la clase sin necesidad de instanciación.

Utilizando un ejemplo algo absurdo para que se entienda: Si en el caso de la clase coche la propiedad color fuera estática, todos los coches serían del color ahí definido, y si lo cambio, cambiaría el color de todos los objetos coche instantáneamente.

Quizá un ejemplo más lógico en relación con el coche sería la propiedad "marca" que sería por ejemplo "Citroën" para todos. Si esta marcar la compra otra empresa, al cambiarla cambiaría en todos los vehículos.

Para definir un miembro como estático se usa la palabra reservada **static**. Veamos un ejemplo en el caso de la clase Perro:

```
static String definicion = "El mejor y más baboso amigo del hombre";
```

Prueba ahora las siguientes líneas en el *main*:

```
System.out.println(perro.definicion);
Perro.definicion= "Ser amigable pluricelular";
System.out.println(perro.definicion);
```

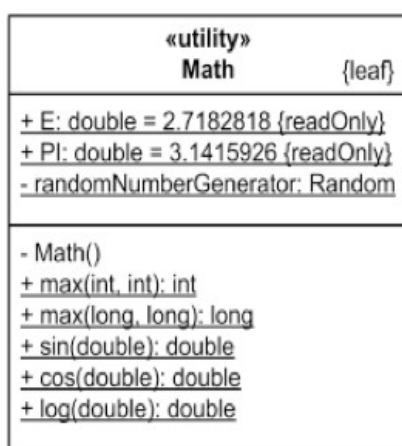
Se debe acceder a los miembros estáticos a través del nombre de la clase, no desde una referencia a uno de los objetos de dicha clase.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

De forma previa a este tema siempre hemos usado la palabra reservada *static* para todas las funciones que creábamos. Esto era porque no instanciábamos ningún objeto y era la única manera de usar funciones sin instanciar nada. Además **el main siempre será estático** pues nadie lo instancia si no que es ejecutado por la máquina virtual.

Una utilidad muy habitual es crear clases con todos sus miembros estáticos para realizar una librería clásica. Un ejemplo que hemos estado usando es la clase Math, que no instanciamos y podemos usar sus miembros.

En UML un miembro estático aparece subrayado en el diagrama de clases. Veamos esto con un ejemplo de la clase Math:



La palabra <<utility>> que aparece en la parte superior es lo que se denomina un “estereotipo”, simplemente es un indicativo que se usa en clases UML para indicar el uso que se le da a esa clase. En el caso de *utility* se refiere a clases que funcionan como librerías clásicas aportando una serie de funciones y atributos de interés (habitualmente constantes o readOnly).

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Objetos como parámetros

Hasta ahora hemos dicho que cuando se pasa una variable por parámetro esta no cambia. El ejemplo lo vimos en el tema anterior cuando cambiábamos el valor de un parámetro en la función pero el valor se conservaba fuera. Veamos que ocurre cuando pasamos objetos como parámetros.

Crea una nueva clase denominada Comida de la siguiente forma:

```
public class Comida{
    public String nombre;
    public double precio;

    public Comida(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }
}
```

Añade a la clase Perro el método regurgitar:

```
public void regurgitar(Comida c){
    c.nombre="bolo alimenticio";
    c.precio =0;
}
```

En el *main* escribe las siguientes líneas:

```
Perro perro = new Perro(5, "Can Palleiro", "Terminator");
Comida pienso=new Comida("pienso",50.25);

System.out.printf("%s %.2f Euros\n",pienso.nombre, pienso.precio);
perro.regurgitar(pienso);
System.out.printf("%s %.2f Euros\n",pienso.nombre, pienso.precio);
```

Al contrario de lo esperado, en este caso sí que cambian los datos ¿A que se debe?

La explicación es que los el parámetro no cambia pues el parámetro es la referencia al objeto (un puntero a memoria) y lo que cambiamos es el objeto. Cuando se pasa un parámetro tipo objeto, lo que se copia es la dirección de memoria en el

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

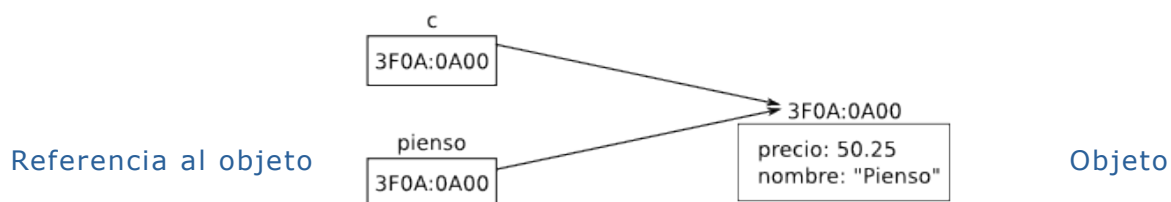
parámetro. Es como si tuviéramos dos variables apuntando a la misma dirección de memoria, al cambiar los datos de esa dirección cambia todo. Puedes verlo de forma similar al concepto de acceso directo a un archivo.

Queda resumido en este esquema:

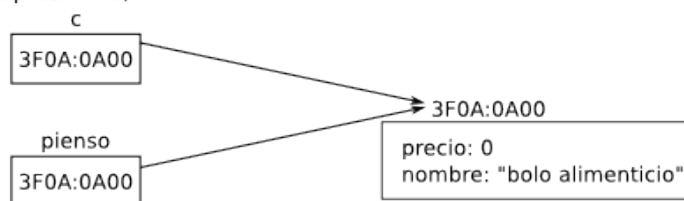
1. Definimos el objeto pienso



2. Llamamos a la función regurgitar
c=comida; (Parámetro = variable)



3. cambia el contenido de c
c.nombre = "bolo alimenticio";
c.precio = 0;



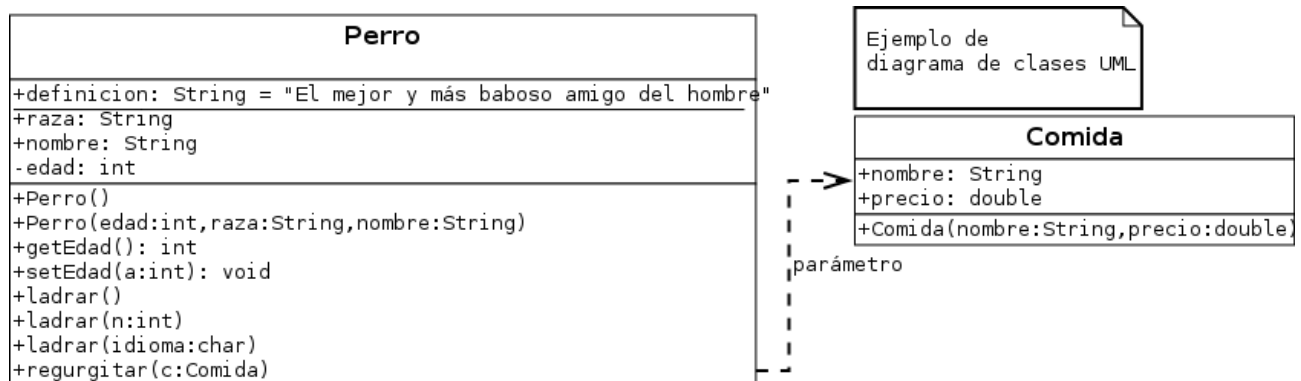
Efectivamente la variable pienso NO CAMBIA sólo cambian los datos apuntados o referenciados por la misma

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Ampliación ED

Diagrama de clases UML

Para finalizar este tema vemos el diagrama UML de clases de lo visto hasta el momento, explicando las novedades.



Podemos ver la clase Perro completa. Cabe notar que el atributo estático *definicion* se escribe subrayado.

Para indicar que existe una dependencia entre la clases Perro y Comida se indica con una línea de trazos terminado en una flecha abierta. En este caso la dependencia es paramétrica ya que Perro depende de Comida pues lo usa como parámetro en uno de sus métodos.

Finalmente el recuadro con una esquina doblada se utiliza para realizar cualquier comentario del diagrama. Se le pueden añadir líneas con flechas para apuntar a una zona concreta del diagrama.

Nota: Existen diversas herramientas para crear diagramas de clases en UML. Programas como Dia, Yed, ArgoUML o Umbrello permiten la realización de dichos diagramas de forma sencilla.

Algunos también permite generar diagramas a partir de código.

Finalmente para distintos IDEs también existen plugins que permiten el desarrollo UML. Por ejemplo en vscode se puede usar la extensión UMLet

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO	Programación					CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR	Francisco Bellas Aláez (Curro)						

Refactorización

Renombrar: En este caso al renombrar en un proyecto (o en un package) una clase, esto hace que cambie en todos los archivos donde se use dicha clase.

Crear campo a partir de variable: Si encontramos necesario que una variable local de un método sea convertido en una propiedad (campo) de la clase, nos situamos encima de dicha variable y con **CTRL+.** y **Extract to field** la convertimos en una propiedad privada de la clase.

Creación automática de **setters/getters**: En la línea de la propiedad que queramos crear

Botón derecho → Source Action → Generate Setters and Getters.

Creación automática de **constructores**: En la clase

Botón derecho → Source Action → Generate Constructors.

Otros snippets interesantes:

- **class**: Clase pública
- **ctor**: Constructor público (con llamada a super, bórrala hasta que sepas qué es).
- **prf**: Propiedad privada (con Tab se van completando el tipo y el nombre).
- **private_method**: Nuevo método privado.

No se recomienda usar las automatizaciones hasta que el programador tenga soltura con todos los conceptos englobados en este tema.