**POO FEBRERO CUADERNO**

Tabla de contenido

[AMP Principios de la POO 3](#_Toc160552160)

[Abstracción 3](#_Toc160552161)

[Encapsulamiento 3](#_Toc160552162)

[Modular 3](#_Toc160552163)

[Herencia 3](#_Toc160552164)

[Polimorfismo 3](#_Toc160552165)

[UML 3](#_Toc160552166)

[¿Qué son las clases, los objetos y las instancias? 3](#_Toc160552167)

[AMP Ventajas de la POO 4](#_Toc160552168)

[Tipos de niveles de acceso 4](#_Toc160552169)

[-Privado 4](#_Toc160552170)

[+Publico 4](#_Toc160552171)

[#Protegido 5](#_Toc160552172)

[AMP Default 5](#_Toc160552173)

[EJEMPLO EN CODIGO JAVA 5](#_Toc160552174)

[Sobrecarga 5](#_Toc160552175)

[TIPOS DE DATOS EN JAVA 6](#_Toc160552176)

[int vs Interger: 7](#_Toc160552177)

[float vs Float: 7](#_Toc160552178)

[true vs True: 7](#_Toc160552179)

[CONSTRUCTORES EN JAVA 8](#_Toc160552180)

[Constructores por defecto 8](#_Toc160552181)

[Constructores con parámetros de otra clase 8](#_Toc160552182)

[TRABAJO DE AMPLIACIÓN 9](#_Toc160552183)

[Relaciones de Asociación 9](#_Toc160552184)

[MULTIPLICIDAD 1.1 11](#_Toc160552185)

[Cambio de multiplicidad 12](#_Toc160552186)

[Asociaciones navegables 12](#_Toc160552187)

[Asociaciones particulares: Composición y agregación 12](#_Toc160552188)

[Relaciones de dependencia 13](#_Toc160552189)

[Diagramas de secuencia 14](#_Toc160552190)

[1º EJERCICIO: Creación de un nuevo sistema, ‘Doors’ que sustituirá a Windows. 15](#_Toc160552191)

[Modificadores Final y Abstract 16](#_Toc160552192)

[SUPERCLASE 16](#_Toc160552193)

[SUBCLASES 16](#_Toc160552194)

[TIPO FINAL 17](#_Toc160552195)

[Método final: 17](#_Toc160552196)

[Variable final: 17](#_Toc160552197)

[Clase final: 17](#_Toc160552198)

[⚠⚠ PROHIBICIÓN EN JAVA ⚠⚠ 18](#_Toc160552199)

[Interfaces (Contratos) 18](#_Toc160552200)

[COMO CREAR Y PROGRAMAR TESTS EN JAVA 19](#_Toc160552201)

[getAlgo(): 20](#_Toc160552202)

[función en el que añadimos una cantidad a una variable ‘algo=0’; sumar(amount): 20](#_Toc160552203)

[función en la que restamos una cantidad a la variable ‘algo’; resta(amount): 20](#_Toc160552204)

[Arrays: 20](#_Toc160552205)

[ARRAYS 20](#_Toc160552206)

[Arrays de clases: 21](#_Toc160552207)

[Errores: 21](#_Toc160552208)

[Importante: 21](#_Toc160552209)

[Matriz(MxN) que recoge las operaciones de los correspondientes M, N: 22](#_Toc160552210)

[Importante: 22](#_Toc160552211)

[UTILIZANDO ATRTIBUTTOSS DE UNA CLASE EN OTRA QUE LA COMPONE 22](#_Toc160552212)

[ERRORES: 23](#_Toc160552213)

[IMPORTANTE: 23](#_Toc160552214)

[STATIC TYPE 23](#_Toc160552215)

[Variable static: 23](#_Toc160552216)

[Método static: 23](#_Toc160552217)

[Bloque static: 23](#_Toc160552218)

[SUPER Y HERENCIA 24](#_Toc160552219)

[POLIMORFISMO 25](#_Toc160552220)

[HERENCIA VS POLIMORFISMO 25](#_Toc160552221)

[TIPOS DE POLIMOORFISMO 25](#_Toc160552222)

[**Estático** 25](#_Toc160552223)

[**Dinámico** 25](#_Toc160552224)

31/1

# AMP Principios de la POO

Abstracción – Prioriza la funcionalidad de un objeto antes que su desarrollo.

Encapsulamiento – El estado de un objeto está oculto a cualquier otro. La única forma de interactuar con él es vía los atributos y métodos que tiene definidos

Modular – Reutilización de código

Herencia – Generalización: Al poder crear una clase a partir de otra, esta última hereda todas sus características y permite añadir otras nuevas

Polimorfismo -Dos objetos distintos de tipos distintos pueden ser tratados de forma general. Podemos enviar mensajes sintácticamente iguales a objetos distintos.

UML – Lenguaje modelado unificado – Describe el proceso completo de ingeniería del software

¿Qué son las clases, los objetos y las instancias? [[1]](#footnote-1)

En la metodología orientada a objetos hay que comenzar definiendo la estructura de los objetos. Imaginemos que queremos crear 2000 casas iguales (con la misma estructura, pero diferentes colores o materiales). En vez de empezar creando cada casa desde cero, primero definiremos un esquema a partir del cual crearemos todas las casas, lo que en POO llamaremos CLASE. Cada una de las casas creadas a partir de ese esquema serán los OBJETOS, tienen una serie de características, llamadas atributos, como podrían ser cantidad de habitaciones, material del que están hechas las casas, barrio en el que viven, habitantes de la casa…, y también engloban una serie de operaciones, denominadas métodos. En el ejemplo de las casas, podríamos realizar ‘operaciones’ como cambio de barrio (si fuese una prefabricada), cambio de habitantes…[[2]](#footnote-2) Se dice que creamos una INSTANCIA de dicha clase cuando creamos un nuevo objeto (una nueva casa)

Resumen

* Las **clases** son las descripciones teóricas de los objetos, incluyen información sobre sus **ATRIBUTOS** (sus datos) y sus **MÉTODOS** (las operaciones que abarcan).
* Los **objetos** son ‘cosas’ que podemos crear y definir, que tienen una serie de características (atributos) y funciones propias (métodos).
* Las **instancias** son procesos de creación de objetos. (Instanciar una clase = Usar la clase)
* Un **evento** es la ejecución de una operación

# AMP Ventajas de la POO

1. La ejecución del programa se reduce a comprender la interfaz de los objetos y la interacción entre ellos.
2. Es posible crear clases a partir de otras clases ya existentes. Promueve la modulación y reduce el número de errores. Se dice entonces que los nuevos objetos heredan todas las características del primero. EJEMPLO: Dentro del sistema del zoológico tenemos un objeto llamado ‘animal’. Cuando queremos crear nuevos objetos como ‘Vaca’ o ‘Ciervo’, estos heredan las características de ‘animal’ y nos permiten añadir atributos y métodos específicos para esos animales.
3. Cuando creamos una clase, podemos reutilizarla en otros programas sin emplear todo el código.
4. Cambios en el código de una clase no afectan al programa entero.
5. El interior de un objeto no es visible al exterior, creando un código más seguro (Sabemos que una parte del código no va a modificar por error los datos utilizados en otra parte del código).
6. Dividir los problemas en partes más pequeñas nos permite testarlas de manera independiente -> más fiabilidad.

# Tipos de niveles de acceso[[3]](#footnote-3)

Son las restricciones que tiene un dato para ser accedido. Se dan en clases y objetos.

## -Privado

Algo (clase, función o variable) está en privado y solo puede ser accesible a los métodos de la misma clase en la que la variable o método se ha declarado.

EJEMPLO

Suponemos que trabajamos dentro de una clase: ‘Persona’:

private int Edad;

La variable Edad solo será visible dentro de la clase, y al intentar acceder a ella:

Persona person;

person = new Persona();

person.Edad = 20

La tercera línea nos dará error al no ser visible.

## +Publico

Algo es público y se puede acceder de manera universal a las clases, variables y los métodos que hemos creado.

Siempre se podrá acceder desde cualquier lugar del código

## #Protegido

Algo está protegido y solo es accesible desde los métodos definidos en las clases que pertenecen al mismo paquete, semejante a private.

Es accesible desde cualquier función de cualquier subclase (las clases heredadas)

Es apropiado para una subclase; no para una clase sin relación

EJEMPLO (continuación)

protected int Edad;

Tenemos el mismo resultado anterior.

Si heredamos la clase Persona, la variable Edad será visible y tratada como privada.

\*si volviéramos a heredar la nueva clase ya no tendríamos acceso.

Los atributos siempre son privados; si se quiere consultar algo, debe ser a través de un método que puede ser de varios tipos de acceso.

## AMP Default

Una variable o método es por defecto default si no tiene ningún nivel de acceso asignado.

Permite acceso desde cualquier método al que pertenecen las clases que sean miembros del mismo paquete en el que esté la variable, función o clase.

\*Son lo que llaman package-friendly o package-private

# EJEMPLO EN CODIGO JAVA

classs coche {

integer CV;

String Matricula;

integer color;

string MarcaModelo;

*public* coche{ (Constructor por defecto; inicialización de variables y cualquier tipo de operación)

}

*public* coche(integer p(arámetro)CV, string pMatricula, integer pColor, string pMarcaModelo){

}

~coche(){

} (destructores, liberan memoria)

Sobrecarga -- No pueden crearse otras funciones con los mismos parámetros en el mismo orden; si estuvieran en un orden diferente sí sería una función nueva

*public* void (void=no devuelve nada) conducir(){}

*public* void aparcar (){}

*protected* integergetCV () {

return this.CV;

}

Ahora crearemos una nueva subclase a partir de la clase coche; ‘SUV’, y le añadiremos 2 nuevos métodos:

-AlturaSuelo integer

+canSubir(integer): boolean

Veamos el código:

class SUV extends Coche{

integer AlturaSuelo;

public SUV (integer pCV, string pMatricula, integer pColor, stringpMarcaModelo; integer pAltura){

super (tírate a por el padre y coge el entero, entero, string…) (pCV, pMatricula, pColor, pMarcaModelo)

this.AlturaSuelo: pAltura

}

public boolean canSubir(integer pS) {return pS <=this.AlturaSuelo}

Class ejemploCoches{ (clase que actúa como lanzador)

public static (static=atributo de la clase que no se puede copiar, singleton) void main (string Aug[]) {

suv micoche = new SUV(200, ‘92929’, 1, ‘Tesla’, 18);

suv tucoche = new SUV(900, ‘82734’, 2, ‘Mercedes Benz’, 17);

if (micoche.canSubir(22))

system.out.print(‘puede subir’);

El punto sirve para llamar a un método o un atributo de una variable siempre que sea publico

# **TIPOS DE DATOS EN JAVA**

1. boolean -> true/false
2. char -> un único carácter : ‘A’
3. byte -> valor que representa un numero de 8-bits. Está contenido en el rango [-128, 127]
4. short -> valor que representa un número de 16 bits. Está contenido en el rango [-32768, 32767]
5. int -> valor que representa un número de 32 bits.
6. float -> valor que representa un número de 32 bits, que permite decimales: 9.99F
7. long -> valor que representa un número de 64 bits: 33422334333L
8. double -> valor que representa un número de 64 bits con capacidad de aceptar decimales.

## int vs Integer:

Integer es una clase que engloba a int pero que le añade algunos métodos. Es una clase creada por el propio java

## float vs Float:

Tiene la misma diferencia que int e Integer. A la hora de escribir un número con decimales, podemos escribirlo con ‘f’ o con ‘F’.

## boolean vs Boolean:

Para empezar, boolean es un tipo de dato primitivo, y Boolean una clase con más métodos.

Para asignar valores, a boolean hay que igualarlo a true o false; a Boolean hay que acceder a su constante: Boolean.TRUE-

Para imponer condiciones, a boolean hay que ponerle el doble == mientras que a Boolean hay que usar el método .equals().

Los valores boolean pueden ser utilizados directamente en código, mientras que para acceder al valor de un Boolean hay que utilizar el método .booleanValue().

Ejemplo

Boolean isMarried = Boolean.TRUE;

Boolean isSingle = Boolean.FALSE;

if (isMarried.equals(Boolean.TRUE) && isSingle.equals(Boolean.FALSE)) {

System.out.println("You are married and not single.");

}

boolean boolValue = isMarried.booleanValue();

if (boolValue && !isSingle.booleanValue()) {

System.out.println("You are married and not single.");

## true vs True:

true hace referencia a un boolean y True a un Boolean

# **CONSTRUCTORES EN JAVA**

Los constructores sirven para inicializar una clase, y nos permiten crear nuevos objetos a partir de una clase.

Pueden depender de unos parámetros (los atributos de la clase) o no depender de nada, como serían los constructores por defecto.

## Constructores por defecto

Los constructores por defecto pueden tener o no parámetros iniciales.

Sí los tienen: algunas clases como círculos o formas, porque necesitamos un radio y un centro inicial para crear el círculo. En este caso, sin parámetros, para establecer el valor de los atributos tan solo hay que poner el valor que queremos:

class Circle{

double radio;

public Circle(){

this.radio = 23;

}

}

## Constructores con parámetros de otra clase

En las relaciones de asociación, como los atributos de una clase1 pueden ser del tipo de una clase2, para inicializarlos, hay que instanciar la clase 2:

class Circle{

double radio;

Punto centro;

public Circle(double radio, int x, int y){

this.radio = radio;

this.centro = new Punto(x,y)

}

}

# TRABAJO DE AMPLIACIÓN

# Diagrama Descripción generada automáticamente**Relaciones de Asociación**

Tabla

Descripción generada automáticamente con confianza media

Diagrama

Descripción generada automáticamente

Existen también relaciones de dependencia y generalización o especialización:

<http://elvex.ugr.es/decsai/java/pdf/3c-relaciones.pdf>

7/02

En las relaciones de asociación, tenemos:

<Rol principal>

|  |
| --- |
| COCHE |
|  |
|  |

|  |
| --- |
| PERMISO CIRCULACION |
|  |
|  |

1.1

0.1

## MULTIPLICIDAD 1.1

Indica que existe exactamente una instancia de tipo de entidad en el extremo de la asociación

EJEMPLO

class multeejo1{

Data rol;

Data rol( =null, (inicializo como 0 porque puede ser vacío, no hay enlace; puede meter una sola instancia de la otra clase)

class data{

public nombre = ‘Antonio’;

multejo1 m;

}

class Multej11{

Dat rol = new Data() (inicializo a un nuevo elemento de manera forzosa; puedo crearlo vacío y en el constructor meterle un dato(el= elemento). Estamos forzando una conexión)

public Multej11(Data el) {

rol = el;

}

class Multej 0n(puedo tener múltiples elementos enlazados con una lista que empieza vacia){

lista rol = new lista();

}

class Multej 1n{

lista rol = new lista();

public Multej 1n (Data el) {

rol.add(el);

}

# **Cambio de multiplicidad**

Consideramos una variante del esquema inicial coche/permiso. Ahora decimos que un coche puede o no tener un permiso de circulación (porque en la vida real es así), la multiplicidad pasa de 1.1 a 0.1. La relación podría ir en ambas direcciones, una, o en ninguna.

# **Asociaciones navegables**

Las asociaciones no tienen por qué ser bidireccionales, podemos hacer que naveguen en una sola dirección, y entonces la línea de asociación concluye en una flecha. En estas asociaciones ‘x’ clase nos lleva a ‘y’ clase.

ROL ALTERNATIVO

0.1

<Rol principal>

|  |
| --- |
| COCHE |
|  |
|  |

|  |
| --- |
| PERMISO CIRCULACION |
|  |
|  |

0.1

# **Asociaciones particulares: Composición y agregación**

Se trata de una relación entre un todo y sus partes.

Se representan con un rombo al final de la línea de asociación.

COMPOSICION

AGREGACION

|  |
| --- |
| FLOTA |
|  |
| Lista de coches |

|  |
| --- |
| COCHE |
|  |
| Lista de ruedas |

|  |
| --- |
| RUEDA |
|  |
|  |

El rombo puede estar relleno o no:

-Relleno: Las partes solo existen asociadas a un compuesto. Una flota no puede existir sin coches.

-Vacío: Las partes pueden existir por separado. Las ruedas pueden estar (o no) puestas en el coche, pero las ruedas siguen existiendo y el coche también.

La diferenciación de estas relaciones es pura INTERPRETACIÓN, porque a nivel de código, son la misma asociación.

# **Relaciones de dependencia**

Texto, Carta

Descripción generada automáticamente

Una instancia de una clase depende de otra.

Se representan con una flecha discontinua.

Los roles de estas asociaciones suelen definirse con verbos como: Usar, importar, redefinir…

EJEMPLO

|  |
| --- |
| TELEFONO |
|  |
|  |

|  |
| --- |
| RED |
|  |
|  |

<USA>



El teléfono depende de la red para funcionar (eficientemente).

No la necesita para funcionar por sí solo, pero la red es lo que le da sentido al teléfono.

# **Diagramas de secuencia**

Plasman la secuencia de operación de algo; describen cómo debería funcionar una secuencia.

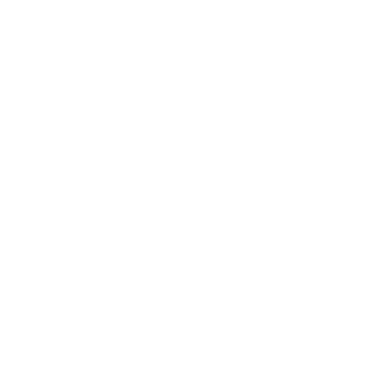
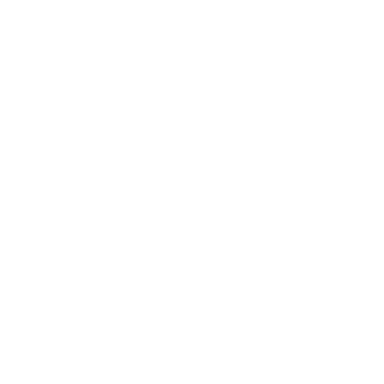
Se crean a partir de los diagramas de clases.

EJEMPLO: Creemos un diagrama de secuencia para la realización de una llamada

|  |
| --- |
| TELEFONO: Antonio |

|  |
| --- |
| RED: Movistar |

|  |
| --- |
| TELEFONO: Maria |



Llamar (A María)

Llamada (María)

Está conectada

SONAR (María)

Descolgar

Aceptar llamada (Aceptada)

Llamada aceptada

HABLAR (María)

Hablar

HABLAR (Antonio)

Hablar

Los rectángulos definen el tiempo de ejecución de una acción.

Podemos realizar una llamada porque existe la clase ‘teléfono’ con ‘x’ métodos; y la clase ‘red’ con ‘y’ métodos.

# **1º EJERCICIO: Creación de un nuevo sistema, ‘Doors’ que sustituirá a Windows.**

Nos hemos cansado de Windows y queremos crear un nuevo sistema. Para ello debemos hacer el diagrama de clases y el de secuencia de un nuevo sistema de ventanas.

Crearemos la clase RECT. Podrá recibir interacciones del ratón y del teclado.

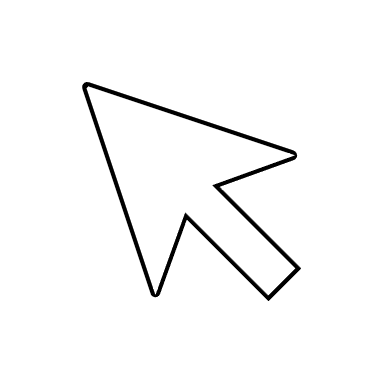
(Px, py)

(0, 0)

|  |
| --- |
| RECT |
| Atributos |
| Px  Py  Dx  Dy |
| Métodos (Reciben eventos[[4]](#footnote-4)) |
| onClick(x,y)  onMove(x,y)  onKeyPress(Key)  Draw\_Border(x, y, dx, dy)  Draw\_pic(x, y, dx, dy, pic)  Draw\_text(x, y, texto)  Draw\_fill(x, y, dx, dy, color) |

Click!

(95, 10)



dy

dx

14/2

# **Modificadores Final y Abstract**

SUPERCLASE (Cosas genéricas; más abstracción) – Vehículo

SUBCLASES (Cosas más concretas, menos abstracción) – SUV, moto, coche, bici

Vehículo

Moto

Bici

Coche

SUV

abstract public class VEHICULO { //No se puede instanciar porque no existe

}

* Si intentamos crear un nuevo objeto:

VEHICULO V= new VEHICULO;

* Nos da error, porque es una clase a la que puedes programar unos métodos que usarán las subclases.

VEHICULO V= new SUV; (del SUV solo puedo llamar los métodos genéricos, los del vehículo)

SUV s = v;

SUV s2 = (SUV)v;

Vehículo

2 ruedas

3 ruedas

4 ruedas

Sin ruedas

**Final**

Nissan

Citroen

SUV

Las clases con modificadores FINAL no permiten que se cree ninguna subclase nueva a partir de estas pero son instanciables; su principal papel es bloquear o detener una jerarquía.

# **TIPO FINAL**

Es un tipo de clase utilizado para indicar que una variable, método, o clase puede ser modificado o sobrecargado.

## Método final:

Un método final no puede ser sobre cargado(no puede haber dos métodos con la misma firma). Y si por ejemplo una super clase tiene un método final, su subclase. no podrá tener ni usar ese método.

## Variable final:

Solo se le puede asignar un valor una vez, después de eso, no se puede cambiar.

Se podrá modificar solo si se trata de un objeto mutable.

## Clase final:

Es un tipo de clase que no puede tener más hijas; ninguna otra clase puede ser extendida de ella.

En resumen, se utiliza para indicar que una variable, método o clase no puede ser sobrecargado, para crear objetos inmutables o asegurarse que el comportamiento de una clase o método no cambia.

# **⚠⚠ PROHIBICIÓN EN JAVA ⚠⚠**

EJEMPLO

Tomemos como ejemplo dos clases: la clase de los peces y la clase de los animales terrestres. Como los anfibios provienen de ambas, podríamos crear un diagrama semejante a este:

ATerrestres

Pez

Anfibios

Lo cual en programación se denomina HERENCIA MÚLTIPLE, pero en java es PROHIBITIVO. (En C++ se usa, pero particularmente en POO no está permitido usar esa herencia, porque puede resultar liosa al no estar las clases *Pez* y *ATerrestres* bien definidas (¿De dónde proceden?))

En vez de eso, usamos interfaces.

# **Interfaces (Contratos)**

Pez

ATerrestres

Implementa

Implementa

Anfibios

En las interfaces definimos la FIRMA DE MÉTODOS PÚBLICOS, no escribimos ni el código ni los atributos (podrían añadirse los atributos públicos, pero no es recomendable)

EJEMPLO

public interface Pez{

public String tipo();

public String toString();

}

public interface Aterrestre {

public int getNumPatas();

}

* **¿Cómo creamos la clase anfibio?**

public class Anfibio implement Pez, Aterrestre { (Implement quiere decir que hay que programarlos)

public String tipo() {}

public String toString() {}

public int getNumPatas() {}

public String otroMetodo() {} //Creamos un nuevo método

}

Ahora la clase anfibio ha heredado los métodos de la clase Pez y ATerrestres, pero le hemos añadido un nuevo método.

Se denomina **contrato** porque el programador solo “promete” que dichos métodos (los de pez y aterrestres) aparecerán en el resultado final (anfibios), no se habla ningún atributo y tampoco se impide añadir nuevos métodos a la clase final.

* **¿Qué sucede si dos interfaces tienen el mismo método?**

Es decir:

|  |
| --- |
| PEZ |
|  |
| +tipoPiel()  +tipo() |

|  |
| --- |
| A.TERRESTRES |
|  |
| +Hábitat()  +tipo() |

En este caso no sucede nada, puesto que nos hemos comprometido que la clase final tendrá el método tipoPiel() y el tipo(), así que ya estará incorporado en anfibios, independientemente de que se repita en ATerrestres o no.

# **COMO CREAR Y PROGRAMAR TESTS EN JAVA**

1. Crear un directorio al mismo nivel que el paquete en el que estemos trabajando
2. Llamaremos al directorio: ‘Tests’
3. Dentro del directorio, creamos otro al que llamaremos ‘java’.
4. Marcamos el directorio de ‘java’ como ‘Test Sources Root’. Ahí es donde se guardarán todos los tests.
5. Crear un nuevo directorio dentro de java, llamado ‘Resources’.
6. Marcar el directorio ‘Resources’ como ‘Test Resources Root’.
7. Ahora entramos dentro de una clase, y pulsamos botón dcho encima del nombre de la clase.
8. Seleccionamos ‘generar Test’, y seleccionamos los méttodos deseados de la clase.

Ahora veremos qué escribir en cada uno de los métodos de la clase “Ejemplo”

## getAlgo():

1. Creamos un nuevo valor para Algo: *tipo* algo = *Cantidad/Cadena*
2. Creamos una instancia de la clase: Ejemplo *ejemploNombre* = new Ejemplo(*v1, v2,* algo)
3. Ahora llamamos al comparador: asserttEquals(*algo*, *ejemploNombre*.getAlgo())

## función en el que añadimos una cantidad a una variable ‘algo=0’; sumar(amount):

1. Creamos una instancia de la clase: Ejemplo *ejemplo2* = new Ejemplo(*v1,v2*,algo)
2. Creamos un nuevo amount: int *cantidad* = 200
3. Y creamos el nuevo balance: int *nuevoBalance* = *ejemplo2*.sumar(cantidad)
4. y llamamos al comparador: assertEquals(*nuevoBalance*, *ejemplo2*.getBalance)

## función en la que restamos una cantidad a la variable ‘algo’; resta(amount):

2. Hay 2 casos:

Cantidad inicial **SUFICIENTE**:

1. Balance inicial al que restar cantidad: int *balanIni* = 800
2. Nueva instancia con dicho balance: Ejemplo *ej3* = new Ejemplo(v1,v2,*balanIni*)
3. Determinamos cantidad a restar: int *cantidad* = 100
4. Cambiamos el balanIni: intt *nuevBalan* = *ej3*.restar(*cantidad*)
5. Y comparamos: assertEquals(*balanIni – cantidad; nuevBalan*)
6. O también: assertEquals(*balanIni – cantida, ej3*.restar(*cantidad*)

Cantidad inicial **INSUFICIENTE**:

1. Balance inicial al que restar cantidad: int *balanIni* = 100
2. Nueva instancia con dicho balance: Ejemplo *ej* = new Ejemplo(v1,v2,*balanIni*)
3. Determinamos cantidad a restar: int *cantidad* = 200
4. Y comparamos: assertEquals(balanIni, ej4.restar(cantidad)

## Arrays:

Para comparar arrays no se puede usar la función asserEquals, sino que hay que usar assertArrayEequals().

# **ARRAYS**

Creación de arrays:

*tipo* [] *nombreArray* = new *tipo*[*num*]

“num” es opocional (puede no haber nada):

si ‘num’ es un número, este indica la cantidad de elementos en el interior del array, y así se asigna elementos al array:

*nombreArray*[0] = 1;

*nombreArray*[1] = 3;

…

*nombreArray*[num-1] = 45;

si no escribimos la cantidad de ítems, podemos escribirlo de la siguiente forma:

int x = 2

int y = 5

*tipo* [] *nombreArray* = new *tipo*[] {x,y}

o simplemente asignarle los valores:

*tipo* [] *nombreArray* = new *tipo*[] {2,5}

## Arrays de clases:

En uno de los ejercicios de la practica 2 nos pedían crear un programa que instancie un array 10 veces de la clase MyPoint, cuyos objetos son un punto de la manera (x,y).

Al principio (sin saber que literalmente había que crear un array de tipo MyPoint), intenté generar las 10 instancias de manera automática. Esta es la forma incorrecta:

for(i = 1, i<11, i++) {

MyPoint ni = new MyPoint(i, i);

}

### Errores:

1. Aunque estaba bien intencionado, el I debe de empezar en 0 cuando usemos el array
2. No se pueden crear instancias de manera automática, por lo que cuando intenté crear varios nombres ‘ni’ y que la i se sustituyera por sí sola no funcionaba.

El resto de las intenciones estaban bien programadas (aunque no funcionen).

Por otra parte, esta sería la forma correcta de crear el array:

MyPoint[] puntos = new MyPoint[10];

for(i=0; i<10; i++) {

puntos[i] = new MyPoint(i+1, i+1)

}

### Importante:

1. Se puede crear un array de cualquier tipo, incluidas clases
2. [10] Es 10 porque se tratan de 10 puntos los que creamos
3. El ‘*for*’ comienza en 0 porque esta vez hablamos de posiciones
4. El i no funciona como nombre, pero sí como posiciones

## Matriz(MxN) que recoge las operaciones de los correspondientes M, N:

En el caso del ejercicio la segunda parte te pedía crear una matriz con todas las distancias entre puntos. Usando el bucle ‘for’, este fue el resultado:

for (int i = 0; i < 10; i++) {  
 for (int j = 0; j < 10; j++) {  
 double distance = points[i].distance(points[j]);  
 System.*out*.printf("%.2f ", distance);  
 }  
 System.*out*.println();  
}

### Importante:

1. Las matrices se crean igual que en Python, con 2 ‘for’
2. Se podría haber puesto points[i].distance(points[j]) en el sout, pero por motivos de organización lo definimos primero como distancia.
3. FORMATO: Como distance es de tipo double, podemos elegir el formato en el que queremos que aparezca. En este caso % = distancia, 2f = nº decimales que muestra.

# **UTILIZANDO ATRTIBUTTOSS DE UNA CLASE EN OTRA QUE LA COMPONE**

En uno de los ejercicios de la práctica 2, de relaciones de composición y agregación nos pedía crear la clase MyLine, que generaba rectas a partir de sus dos atributos; un punto de comienzo y otro de final. Los puntos a su vez estaban generados por unas coordenadas sacadas de los atributos de MyPoint.

Para instanciar la clase MyLine primero te pedía crear un constructor, pero utilizaba como parámetros los atributos de la clase MyPoint. He aquí cómo hacerlo mal y bien:

MAL:

public MyLine(int x1, int y1, int x2, int y2) {  
 MyPoint.X = x1;  
 this.begin = x1,y1;

}

Al principio intenté acceder a las coordenadas X de la clase MyPoint, pero no pude porque son privadas y además no se podrían diferenciar x1 de x2.

También intenté acceder a las coordenadas iniciales x1,y1 del atributo ‘begin’; pero tampoco se podría porque ‘begin’ es un objeto de tipo MyPoint, y x1 y1 son valores int primitivos. No se puede asignar un int a una variable de un objeto así porque sí.

BIEN:

Para asignarlo correctamente, habría que asignar el ‘begin’ y el ‘end’ a un nuevo objeto de MyPoint:

public MyPoint(int x1, int y1, int x2, int y2) {

this.begin = new MyPoint(x1, y1);

this.end = new MyPoint(x2,y2);

}

## ERRORES:

1. No se puede acceder a los atributos de una clase (porque están privados) y además siempre se accede a los atributos de un OBJETO de la clase, no a la clase en sí.
2. No se puede asignar una variable int a una variable de un objeto así porque sí.

## IMPORTANTE:

1. Solo podemos acceder a los atributos de una instancia de la clase; no podemos acceder ni modificar a la clase directamente.
2. Begin y End son OBJETOS de MyPoint.

# **STATIC TYPE**

El tipo estático indica que una variable, método o bloque pertenece a una clase y no a una instancia

## Variable static:

Es una variable que pertenece a una clase en vez de a una instancia. Solo hay 1 copia de la variable estática que se comparte por todas las instancias de la clase.

## Método static:

Es un método que pertenece a la clase global y no a instancias. Significa que el método estático puede ser llamado usando el nombre de la clase y no una instancia.

Los métodos static no pueden acceder a variables de instancias o métodos y necesitan usar una instancia de la clase para acceder.

## Bloque static:

Un bloque de código es estático cuando se ejecuta una sola vez. Se suelen usar para inicializar variables estáticas.

# **SUPER Y HERENCIA**

Las relaciones de herencia se basan en que a partir de una clase madre nacen varias clases hijas que se apoderan de todos los atributos y métodos de la clase madre.

A la hora de programar la clase hija, se utiliza la palabra ‘super’ como método para acceder a la clase madre.

Para llamar al Constructor:

Dada una superclase con su propio constructor; tenemos una subclase con sus propios atributos donde queremos crear un constructor con parámetros de la subclase y de la superclase. A la hora de asignar valores de los parámetros de la super clase, usamos el super:

public class General{

atrib1;

atrib2;

}

public class Subegeneral extends General{

atrib3;

public Subgeneral(atrib1, atrib2, atrib3){

super(atrib1);

super(atrib2);

this.atrib3 = atrib3;

}

}

Para llamar a los Miembros de la superclase cuando también existen en la subclase:

Miembros pueden ser variables y métodos.

Al Código de antes le añadimos el método getAtributo1 a la superclase.

Ahora en la función toString accedemos al método previo:

public String toString(){

return “Este texto quiere devolver el atributo1:” + super.getAtributo1

}

Para Sobrescribir métodos:

Cuando se quiere sobrescribir un método de la superclase; extender el comportamiento del método en la subclase sin perder funcionalidad original.

Es decir, sobrecargar un método e implementarle un código diferente. (Polimorfismo)

# **POLIMORFISMO[[5]](#footnote-5)**

Cualidad por la que un objeto puede adquirir distintas formas. Concretamente, en java es la creación y uso de dos métodos con mismo nombre y diferente.

Se utiliza generalmente para que dos funciones con misma finalidad acepten parámetros diferentes.

Se recomienda usar el mismo nombre por el sistema abierto-cerrado; abierto a la extensión y cerrado a la modificación.

## HERENCIA VS POLIMORFISMO

En la herencia, las clases hijas heredan todos los métodos y atributos de la clase padre, sin modificar los métodos de la clase padre. Herencia: reutiliza código.

En el polimorfismo, los atributos son iguales pero los métodos(se llaman igual) tienen su propia implementación. Polimorfismo: dinamiza el código y simplifica funciones. Según qué parámetros se introduzcan, se usará una función u otra.

## TIPOS DE POLIMOORFISMO

**Estático** (de compilación):

Se usa para crear varios métodos con el mismo nombre en la misma clase con parámetros de tipos distintos.

**Dinámico** (de ejecución):

Se usa para que un método de la clase padre tenga la misma finalidad en la clase hija pero con un código propio.

Ejemplo:

A partir de la superclase Shape con su método vacío getArea, creamos 2 subclases: círculo, cuadrado. Las dos tendrán el método getArea(), pero cada una tendrá un código de cálculo diferente.

1. <https://interactivechaos.com/es/manual/tutorial-de-python/clases-e-instancias> [↑](#footnote-ref-1)
2. Nota: En el uso de clases y objetos, no debemos de preocuparnos del desarrollo de sus métodos, solo nos importa cómo se encuentra definida la interfaz del objeto y su funcionalidad. URL: <https://interactivechaos.com/es/manual/tutorial-de-python/atributos-y-metodos> [↑](#footnote-ref-2)
3. <https://lenguajesdeprogramacion.net/diccionario/diferencia-entre-public-protected-o-private/> [↑](#footnote-ref-3)
4. Evento: ejecución de una opoeración [↑](#footnote-ref-4)
5. <https://blog.hubspot.es/website/polimorfismo-java> [↑](#footnote-ref-5)