
epam
Java Básico
Christian Velazquez
Enero, 2017

Qué es Programación Orientada a Objetos (POO)?

Es una **abstracción** del mundo real que representa objetos reales y sus interacciones

Qué es un objeto?

Son una representación de objetos del mundo real que tienen dos características: **estado** y **comportamiento**.

Si miramos a nuestro alrededor, cada cosa que veamos será un objeto que tendrá estas características y que puede ser representado usando el paradigma de POO.

Qué es un objeto?

Estado:
- Nombre
- Color
- Edad

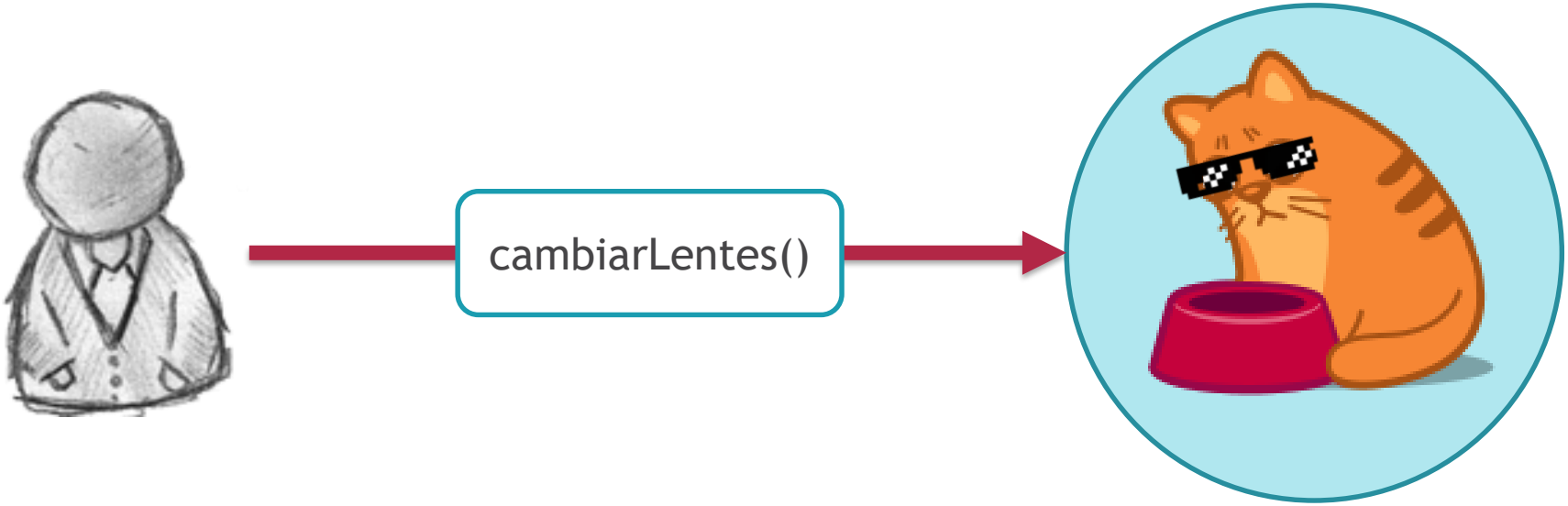
Fields

Comportamiento:
- Comer
- Maullar

Methods

Encapsulamiento

El encapsulamiento ocurre cuando ocultamos el estado interno de una clase y cuando nos aseguramos de que toda interacción del objeto con el mundo exterior se haga a través de sus métodos.



Qué beneficios tiene?

- Modularidad y mantenibilidad
- Ocultamiento de la información
- Reutilización de código
- Extensibilidad

Qué es una clase?

Es un prototipo (blueprint) a partir del cuál se crean objetos “de un mismo tipo”.

Una clase establece el estado y comportamiento base que tendrán todos los objetos que sean de ese tipo.

Todos los gatos tienen un color.

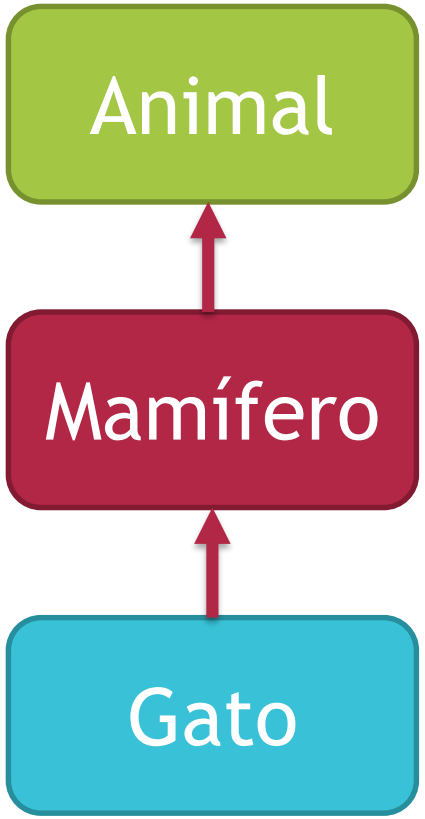
Todos los gatos maúllan.

Herencia

Objetos de diferente tipo pueden compartir características comunes.

De igual manera, cada uno de estos objetos puede tener características individuales adicionales.

En POO, las clases pueden *heredar* estado y comportamiento de otras clases.



Interfaz

Es un contrato para garantizar el comportamiento de una clase con el mundo exterior.

Clases abstractas

Las clases abstractas son clases que pueden contener métodos abstractos y concretos y cuyo único propósito es el de heredarse (no pueden instanciarse directamente).

IS-A vs HAS-A

IS-A es una relación que se basa en la herencia de clases y en la implementación de interfaces.

Si una clase hereda de otra o implementa una interfaz, se dice que esa clase **es de** ese tipo.

Por ejemplo, *un gato **es un (IS-A)** mamífero*.

IS-A vs HAS-A

HAS-A es una relación que se basa en el uso (composición), en lugar de la herencia.

Existe una relación HAS-A cuando una clase tiene una referencia a una instancia de otra clase.

Por ejemplo, *Un gato **tiene un (HAS-A)** collar.*

Polimorfismo

Polimorfismo significa muchas formas. Se considera que un objeto es polimórfico cuando pasa una prueba IS-A para dos o más tipos.

Por ejemplo,

- un gato es un gato
- un gato es un mamífero
- un gato es un animal

Polimorfismo

La manera más común de polimorfismo en Java es referenciar a un objeto por cualquier de sus superclases o las interfaces que implementa.

Composición

Composición se basa en el uso de unas clases por otras.

Composición permite que las clases y su comportamiento se construyan a partir de utilizar otras clases y delegarles tareas específicas.

Por ejemplo, *Un automóvil **tiene** un motor y ruedas*.

<epam>

Alguna pregunta?

Antes de continuar...

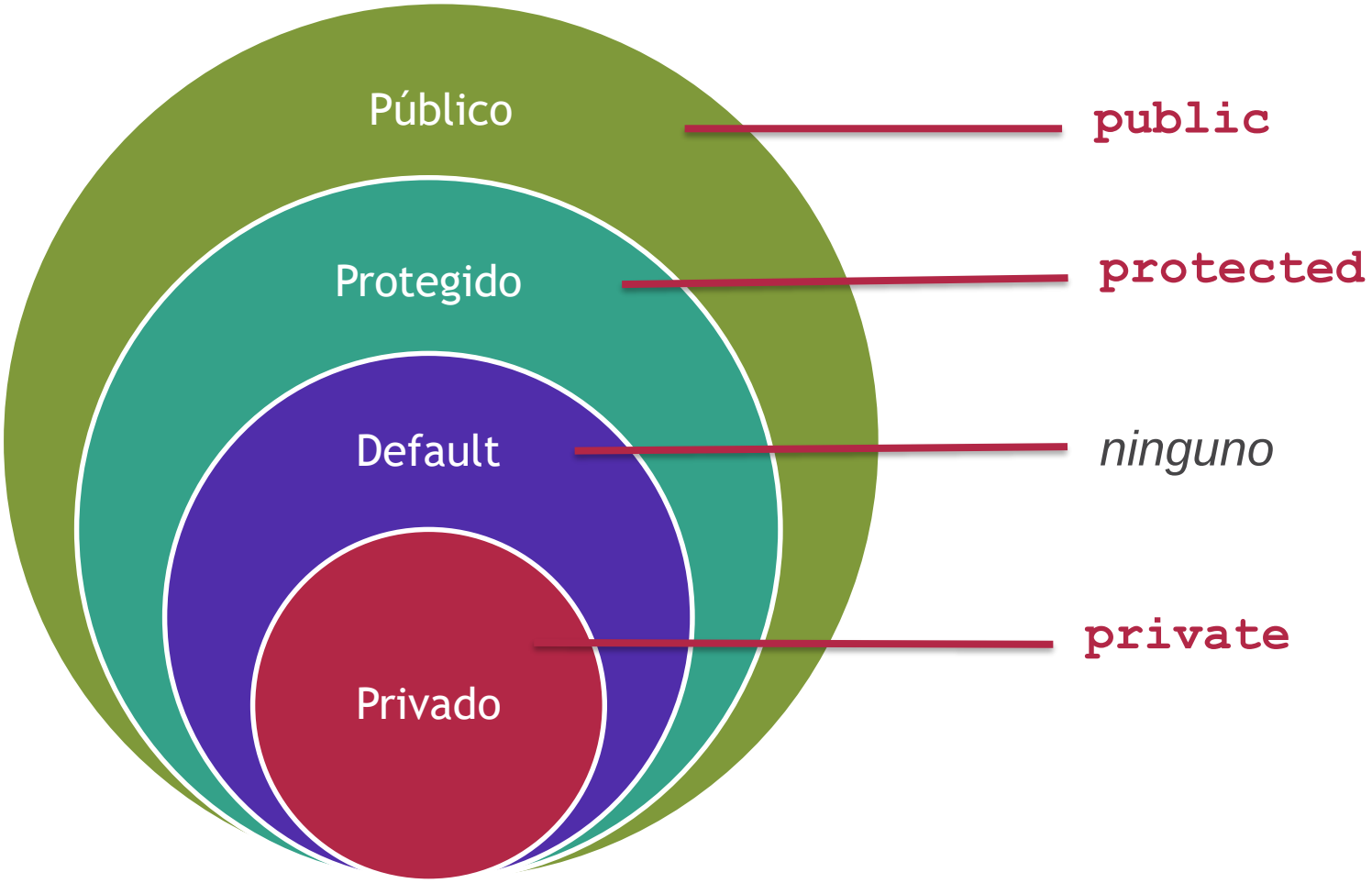
Creemos el clásico "Hola Mundo" en Java.

El código debe ser guardado en un archivo con extesión **.java**

El comando para compilar un archivo java es **javac <Nombre del archivo>**

El comando para ejecutar una clase java es **java <Nombre de la clase>**

Niveles de acceso



Niveles de acceso

Nivel de acceso	Modificador	Clase	Paquete	Subclases	Toda la aplicación
Público	public	S	S	S	S
Protegido	protected	S	S	Por Herencia	N
Default	*ninguno*	S	S	N	N
Privado	private	S	N	N	N

Otros modificadores

final

Declaración de clase.

*Una clase marcada como **final** no puede ser heredada.*

A pesar de que va contra uno de los pilares de la POO, que es la herencia, en ocasiones es necesario asegurar que una clase no puede especializarse, por ejemplo **String**.

Si pudiéramos extender la clase String y proveer nuestra propia subclase donde un String es esperado, podrían suceder cosas inesperadas.

Otros modificadores

final

Declaración de método.

*Un método marcado como **final** no puede ser **sobreescrito**.*

Mientras la subclase hereda el método, no podemos redefinir el comportamiento de dicho método en la subclase.

Esto asegura que el método se va a comportar como se estableció en la clase padre.

Otros modificadores

final

Declaración de variable.

*Una variable marcada como **final** no puede cambiar su valor una vez que éste ha sido inicializado.*

Para tipos primitivos como **int** o **char** esto significa que una vez dado el valor, este no cambia en el futuro.

Para variables de referencia como **Object** o **Cat** esto significa que dicha variable no puede hacer referencia a ningún otro objeto en el futuro.

Otros modificadores

static

*Los miembros de una clase (variables y métodos) marcados como **static** pertenecen a la clase y no a un objeto en particular.*

Esto significa que todas las instancias de dicha clase (todos los objetos generados a partir de dicho prototipo) tendrán una referencia a la misma variable y los cambios sobre ésta se verán reflejados para todos los objetos.

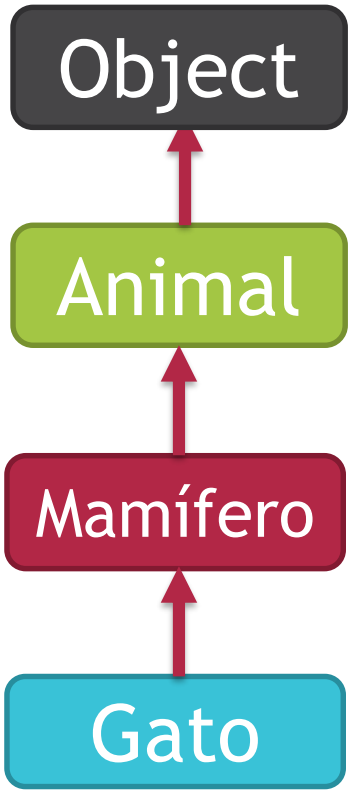
Constructores y bloques de inicialización

Los objetos se construyen. Todas las clases, incluyendo las abstractas, tienen constructores.

Los constructores deben llamarse igual que su clase, pueden tener parámetros y no deben tener ningún valor de retorno.

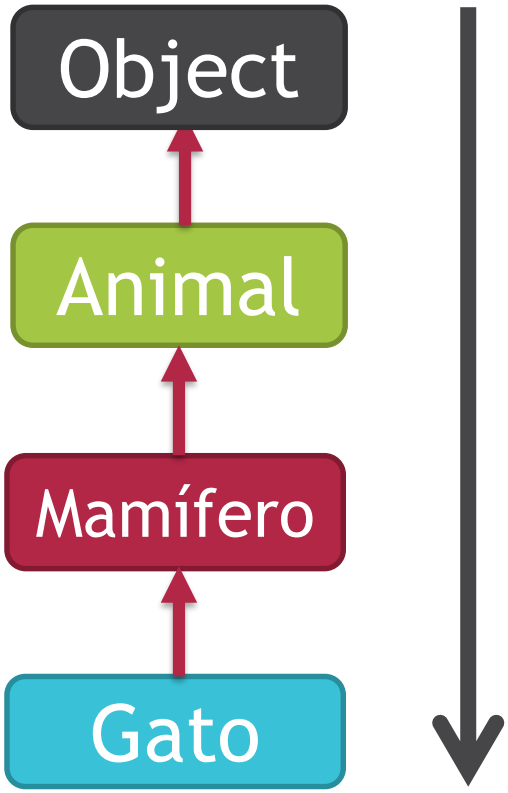
Si no creamos un constructor explícitamente, el compilador agregará uno sin parámetros.

Constructores y bloques de inicialización



7. Constructor de Object se ejecuta
6. Constructor de Animal llama super()
5. Llamada al constructor de Animal
4. Constructor de Mamífero llama super()
3. Llamada al constructor de Mamífero
2. Constructor de Gato llama super()
1. Llamada al constructor de Gato

Constructores y bloques de inicialización



8. Constructor de Object se completa y regresa el llamado al constructor de Animal.

9. Constructor de Animal se completa y regresa el llamado al constructor de Mamífero.

10. Constructor de Animal se completa y regresa el llamado al constructor de Gato.

11. Constructor de Gato se completa.

Constructores y bloques de inicialización

Los bloques de inicialización son lugares donde podemos colocar código que será ejecutado cuando la clase sea cargada o cuando creemos una instancia de la clase.

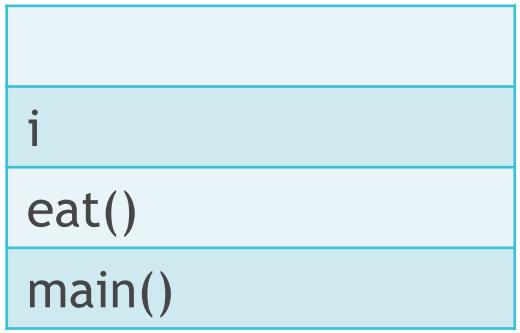
Pueden ser de dos tipos:

Estáticos. Se ejecutan una única vez cuando la clase es cargada
De instancia. Se ejecutan cada vez que creamos una instancia de una clase.

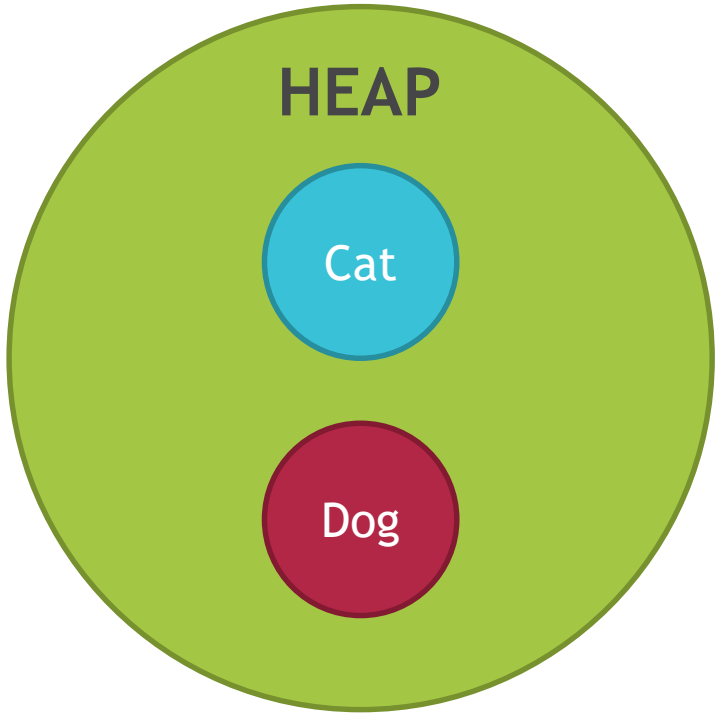
Puede haber N bloques de inicialización y se ejecutan en el orden en que aparecen.

Dónde viven las variables, métodos y objetos?

Pueden vivir en dos lugares, **el heap** y **el stack.**



Métodos y variables locales viven en el stack



Variables de instancia y objetos viven en el heap

Tipos de variables

Primitivos.

- char
- boolean
- byte
- short
- int
- long
- double
- float

Variables de referencia. Para hacer referencia a un objeto.

<epam>

Tipos de variables - Primitivos

Tipo	Bytes
byte	1
short	2
int	4
long	8
float	4
double	8
boolean	NA
char	2

<epam>

Tipos de variables – De referencia

Se refiere a aquellas variables que en lugar de almacenar un valor, contienen una referencia a un objeto en el heap.

Por ejemplo,

```
LandTransport landTransport = new LandTransport();
```

Literales para primitivos

Tipo	Literal
Enteros (byte, short, char, int, long)	decimal, octal, hexadecimal, binario
Flotantes (float, double)	Números enteros y decimales
Boleanos (boolean)	true, false

Literales para String

Los valores literales para String son muy comunes y se especifican entre comillas dobles (“). Ciertos caracteres deben ser “escapados” dentro del literal para que se muestren correctamente, como comillas (\”) o saltos de línea (\n).

Por ejemplo,

```
String helloWorld = "Hello \nWorld";
```

Scopes

Las variables pueden ser declaradas en diferentes partes de una clase. Dependiendo de donde sean declaradas dependerá el “tiempo de vida” y el acceso que se tenga a ellas.

Los scopes posibles de las variables son:

- Estático
- Instancia
- Locales (incluyendo bloques de inicialización y constructores)
- Bloque

Operadores

Los operadores nos permiten obtener nuevos valores a partir de dos o más operandos.

Existen diferentes tipos:

- De asignación (=, +=, -=)
- Relacionales (<, <=, >, >=, ==, !=).
- Aritméticos (+, -, *, /, %)
- Condicional (ternario)
- Lógicos (&&, ||, &, |, !, ^)
- Incrementos (++) y decrementos (--)
- instance of

Control de flujo

Java, al igual que otros lenguajes, ofrece sentencias que nos permiten controlar el flujo que sigue la ejecución de nuestro programa, dependiendo de ciertas condiciones que nosotros establezcamos.

<epam>

Control de flujo

if-else

Una sentencia if permite probar si una expresión es verdadera o falsa, es por ello que dicha expresión debe ser boleana.

Si la expresión se cumple, se ejecuta el código contenido en el bloque **if**.

Podemos probar más de una condición utilizando **else if** por cada condición adicional y podemos indicar una sección de código que se ejecutará si ninguna de las previas es verdadera con la palabra reservada **else**.

Veamos un ejemplo…

Control de flujo

switch

switch se encarga de probar una variable/expresión contra diferentes valores posibles.

La estructura básica de un switch es la siguiente:

```
switch(expresión){
    case constant1: code block
    case constant2: code block
    default: code block
}
```

Veamos un ejemplo…

Control de flujo - Loops

Los loops o ciclos son estructuras que se encargan de repetir un bloque de código mientras una condición se cumpla.

En Java existen tres diferentes: **while**, **do** y **for**.

<epam>

Control de flujo

while

Se utiliza cuando no sabemos cuantas veces vamos a ejecutar un bloque de código, simplemente sabemos que se ejecutará mientras una condición sea verdadera.

Veamos un ejemplo…

Control de flujo

do

Este tipo de ciclo es similar a **while**, la diferencia radica en que **while** evalúa la expresión primero, si se cumple ejecuta el bloque de código.

do ejecuta el bloque de código al menos una vez, es decir, no evalúa la expresión primero. Después de la primera ejecución procederá a evaluar la condición y continuar hasta que ésta no se cumpla.

Veamos un ejemplo…

<epam>

Control de flujo

for

for se utiliza cuando sabemos de antemano cuantas veces ejecutaremos un bloque de código.

Existen dos formas del ciclo **for**.

La básica:

```
for(inicialización; condición; iteración){
    bloque de código
}
```

<epam>

Control de flujo

for

Y una creada para iterar sobre colecciones y arreglos:

```
for(declaración : expresión){
      bloque de código
}
```

Veamos ejemplos de ambas…

Control de flujo – break y continue

Existen ocasiones donde es necesario salir de los ciclos antes de que terminen su ejecución normal o incluso continuar la siguiente iteración sin concluir la actual.

break y **continue** son palabras reservadas que nos permiten tener este tipo de influencia sobre los ciclos.

Control de flujo – break y continue

break nos permite terminar la ejecución de un ciclo.

continue nos permite terminar la iteración actual e intentar continuar con la siguiente. El que continúe o no depende de si la condición del ciclo sigue siendo verdadera.

Control de flujo – labeled statements

También es posible que tengamos ciclos anidados, esto es, un ciclo dentro de otro.

Al utilizar **break** y **continue**, éstos actúan sobre el ciclo actual, si quisiéramos que actuaran sobre un ciclo externo podemos utilizar algo conocido como *labeled statements*, para referirnos a un ciclo en particular por medio de una etiqueta.

```
outer:while(true){
      // Codigo
}
```

Veamos un ejemplo…