**Academia Java – JAVA**

***Vida y Muerte de un Objeto***

Tabla de contenido

[1. No olvides esto cuando diseñas una jerarquía como esta 3](#_Toc313542550)

[1.1 Como es un objeto resultado de new Animal()? 5](#_Toc313542551)

[1.3 El compilador no te permitirá instanciar una clase abstracta 7](#_Toc313542552)

[1.4 Abstracta vs Concreta 8](#_Toc313542553)

[1.5 Cada clase en Java hereda de Object 11](#_Toc313542554)

[1.6 Que pasa si necesitamos cambiar el contrato? 17](#_Toc313542555)

[1.7 Interfaces al Rescate 20](#_Toc313542556)

[1.8 Como sabes cuando hacer una clase, subclase, una clase abstracta o una interface 23](#_Toc313542557)

[1.9 Usando super 23](#_Toc313542558)

# Vida y muerte de un Objeto

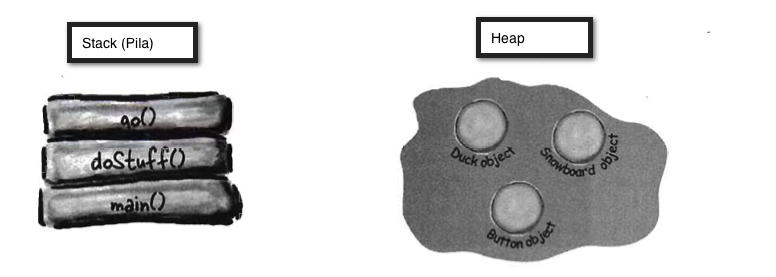
Los Objetos nacen y mueren. Nosotros estamos a cargo del ciclo de vida un objeto. Nosotros decidimos cuando y como construirlos. Nosotros decidimos cuando destruirlos. Claro esta, que nosotros realmente no lo destruimos por nosotros mismos, simplemente los abandonamos a su suerte. Una vez abandonados el Garbage Collector (gc) o Recolector de Basura los hace polvo, reclamando la memoria que el objeto esta utilizando.

El programador Java tiene que saber como se crean, viven, y abandonan objetos eficientemente.

Por eso en esta guía hablaremos del heap, stack (pila), scope (alcance), constructores, super constructores, referencias nulas, y mas.

## 1.2 La pila y el heap: donde las cosas viven

En Java hay dos áreas de memoria donde todo vive por un periodo de tiempo. Los objetos viven en el heap, y lo demás como métodos invocados, variables locales viven en la pila (stack). El uso de la memoria depende del JVM y generalmente no tenemos nada que decir ahí.



**Variables de instancia**

Las variables de instancia son declaradas al interior de una clase pero no al interior de un método.

Ellos representan los “campos” que cada objeto individual tendrá (los cuales pueden ser llenados con diferentes valores por cada instancia de la clase). Las variables de instancia viven dentro del objeto al que pertenecen.

public class Pato {

Cada Pato tendrá una variable de instancia tamaño.

int size;

}

**Variables locales**

Las variables locales son declaradas en el interior de un método, incluyendo parámetros de un método. Estos son temporales, y viven solamente el tiempo de vida del método en la pila.

i y b son variables locales.

public void foo(int x) {

int i = x + 3;

boolean b = true;

}

**Los Metodos son apilados**

Cuando tu invocas a un método, el método se añade en el top de la pila. Se guarda el estado del método, incluyendo la línea de código que se esta ejecutando y los valores de todas las variables locales.

El método que esta en el top de la pila es el método actual que se esta ejecutando.

Un método permanece en la pila hasta que se ejecute completamente. Si el método foo() llama al método bar(), el método bar() es apilado sobre foo().



Veamos el siguiente ejemplo:

public void doStuff() {

boolean b = true;

go(4);

}

public void go(int x) {

int z = x + 24;

crazy();

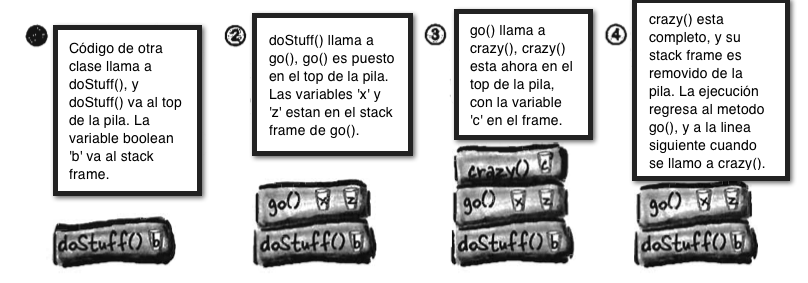
//mas codigo aquí

}

public void crazy() {

char c = ‘a’;

}



**Que hay acerca de las variables locales que son objetos?**

Recuerda, una variable no primitiva guarda una referencia a un objeto, no el objeto en si. Sabemos que los objetos viven en el heap. No importa desde donde se han declarado o creado. **Si la variable local es una referencia a un objeto, solo la variable (la referencia / control remoto) va a la pila.**

El objeto si va al heap.

public class StackRef {

public void foof() {

barf();

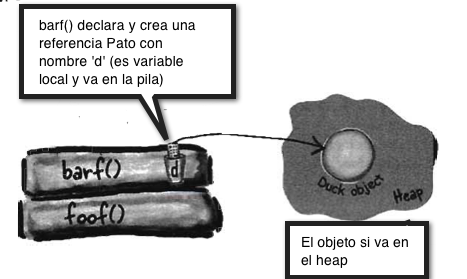
}

public void barf() {

Pato d = new Pato(24);

}

}



|  |  |
| --- | --- |
| * Java tiene dos areas de memoria: la pila y el heap * Las variables de instancia son declaradas al interior de una clase pero fuera de un metodo * Las variables locales son variables declaradas al interior de un metodo o parametro de un metodo * Todas las variables locales viven en la pila, en el frame correspondiente al metodo donde las variables son declaradas. * Las variables de referencia objetos trabajan como variables primitivas – si la referencia es declarada como una variable local, esta va en la pila. * Todos los objetos viven en el heap, sin importar si la referencia es una variable local o variable de instancia | C:\Users\ingas\Desktop\Atencion2.png |

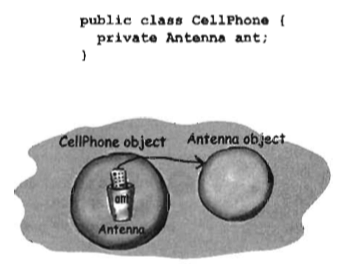
**Si las variables locales viven en la pila, donde las variables de instancia viven?**

Cuando decimos new CellPhone(), Java tiene que hacer espacio en el Heap para ese CellPhone. Pero cuanto espacio? Lo necesario o suficiente para el objeto, lo que significa suficiente “casa” para todas las variables de instancia. Eso es, **las variables de instancia viven en el heap,** al interior del objeto al cual pertenecen.



Recuerda que los valores de una variable de instancia viven al interior de un objeto. Si las variables de instancia son todas primitivas, Java hace espacio para las variables de instancia basado en el tipo del primitivo. Un int necesita 32 bits, un long 64 bits, etc. Java toma cuidado de todo el tamaño, no importa el valor, sea 32 o 32,000000.

Pero, que pasa si las variables de instancia con objetos? Que pasa si un CellPhone tiene una Antenna? En otras palabras tiene una variable de referencia de tipo Antenna.

****

Cuando el nuevo objeto tiene variables de instancia que son referencias a objetos en lugar de primitivos nos viene esta pregunta: El objeto necesitara espacio para todos los objetos que este referencia? La respuesta es, no exactamente. Java reserva espacio para todas las variables de instancia, pero, recordemos que las variables de referencia no guardan el objeto en si, sino son control remoto del objeto. Asi si CellPhone tiene una variable de instancia declarada como tipo no-primitivo Antenna, Java hace espacio dentro de CellPhone solo para el control remoto a Antenna, pero, no para el objeto Antenna.

Cuando el objeto Antenna obtiene espacio en el heap? Primero depende de cuando el objeto Antenna es creado. Si la variable de instancia se declara, pero, ningún objeto se asigna, solo el espacio para la variable de referencia es creada.

**private Antenna ant;**

No hay objeto Antenna actual creado en el heap hasta que se asigne un nuevo objeto Antenna.

**private Antenna ant = new Antenna();**

# Constructor

Cuando creamos un objeto

Pato pato = new Pato(); // Luce como un método por tener los paréntesis

**Pero, no es un método, es un constructor.**

Un constructor luce como un método, pero, no es un método. Este ejecuta el código cuando dices new o en otras palabras cuando instancias un objeto.

La única forma de llamar a un constructor es con el keyword new seguido por el nombre de la clase. El JVM encuentra esa clase e invoca el constructor de dicha clase.

Aunque también cabe aclarar que se puede llamar a un constructor desde el interior de otro constructor.

**Donde esta el constructor? Si no hemos escrito este. Quien lo ha hecho?**

Tu puedes escribir un constructor para tu clase, pero, si no lo haces, **el compilador escribe uno para ti.**

Así es como un constructor por defecto luce:

public Pato() {

}

**Observas que falta algo? Observas la diferencia con un método?**

public Pato() {

}

No hay tipo de retorno.

**Construyendo un Pato**

La característica clave de un constructor es que este se ejecuta antes de que el objeto sea asignado a una referencia. Esto significa que tenemos una chance para configurar cosas en el antes de un objeto sea utilizado. En otras palabras, antes, de que cualquiera use el control remoto para un objeto. En nuestro constructor de Pato, no haremos nada útil, pero demostraremos la secuencia de eventos.

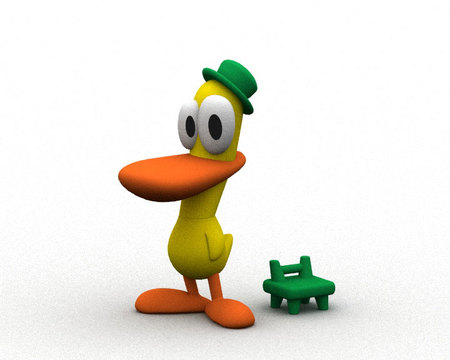
public class Pato {

public Pato() {

System.out.println(“Quack Quack”);

}

}



public class UsarPato {

public static void main (String[] args) {

Pato p = new Pato();

Este llama al constructor.

}

}

## 2.1 Inicializando el estado de un nuevo Pato

Muchas personas usan constructores para inicializar el estado de un objeto. En otras palabras para asignarle valores a las variables de instancia de un objeto.

public class Pato {

int size;

public Pato() {

System.out.println(“Quack Quack”);

}

public void setSize(int newSize) {

size = newSize;

}

}

public class UsarPato {

Pato p = new Pato();

p.setSize(42);

}

* Las variables de instancia tienen valor por defecto. 0 o 0.0 para números primitivos, false para booleanos y null para referencias.

## 2.2 Usando el constructor para inicializar el estado del Pato

El mejor lugar para colocar código de inicialización es en el constructor. Todo lo que necesitamos es hacer un constructor con argumentos.

public class Pato {

int size;

public Pato(int duckSize) {

System.out.println(“Quack”);

size = duckSize;

System.out.println(“El size es ” + size);

}

}

public class UsarPato {

public static void main (String[] args) {

Pato p = new Pato(42);

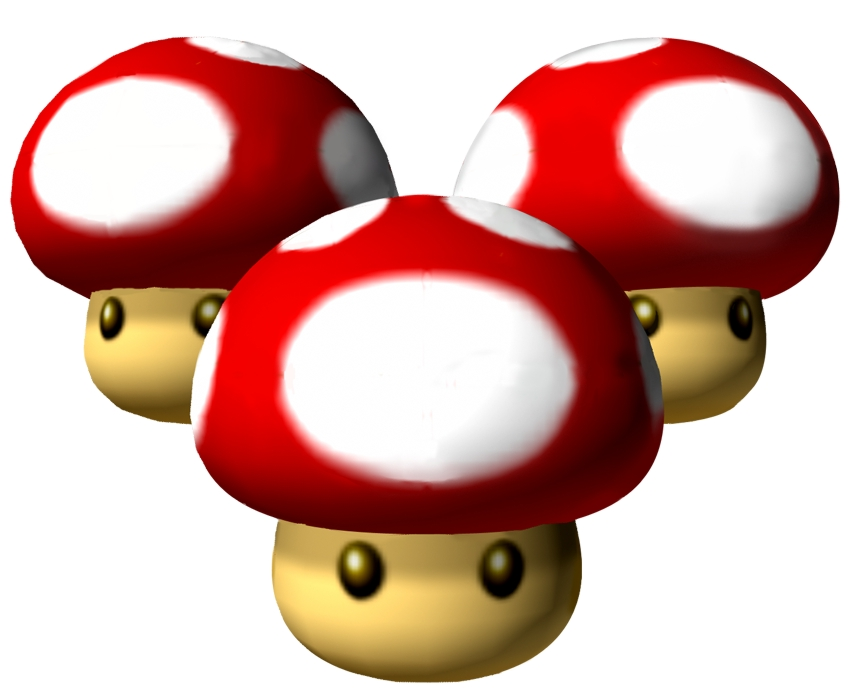
}

}

**ATENCION A LO SIGUIENTE: Si declaras un constructor con argumentos, el constructor sin argumentos lo tendrás si y solamente si lo declaras. El compilador ya no nos lo dará por defecto.**

## 2.3 Constructores sobrecargados significa que tienes mas de un constructor en tu clase

**Para compilar, cada constructor debe tener una lista diferente de argumentos.**

****

public class Hongo {

public Hongo(int size) {}

public Hongo() {}

public Hongo(boolean esMagico) {}

public Hongo(boolean esMagico, int size) {}

public Hongo(int size, boolean esMagico) {}

}

|  |  |
| --- | --- |
| * Las variables de instancia dentro del objeto pertenecen al Heap * Si la variable de instancia es una referencia a un objeto, ambos la referencia y el objeto estan en el heap. * Un constructor es el codigo que se ejecuta cuando tu dices new en un tipo de clase * Un constructor debe tener el mismo nombre de la clase y no debe tener un tipo de retorno * Puedes usar un constructor para inicializar el estado (las variables de instancia) del objeto que esta siendo construido * Si no pones un constructor en tu clase, el compilador te dara un constructor por defecto * El constructor por defecto es siempre sin argumentos * Si colocas un constructor, cualquier constructor, en tu clase, el compilador no te dara el constructor por defecto * En otras palabras, si ya tienes un constructor con argumentos y deseas uno sin argumentos, tendras que ponerlo por ti mismo * Siempre provee un constructor sin argumentos si tu puedes, para hacer facil la vida a los programadores para trabajar con el objeto. * Constructores sobrecargados significa que tienes varios constructores * No puedes tener dos constructores con la misma lista de argumentos. Una lista de argumentos incluye el order y/o tipo de argumentos * Las variables de instancia tienen valores por defecto aun cuando no le des un valor. 0,0.0 para primitivos, false para boolean, null para referencias. | C:\Users\ingas\Desktop\Atencion2.png |
|  |  |

## 2.4 Hacer un Hipopotamo significa que haremos el Animal y Object tambien…

public class Animal {

public Animal () {

System.out.println(“Hacer un Animal”);

}

}

public class Hipopotamo extends Animal {

public Hipopotamos () {

System.out.println(“Hacer un Hipopotamo”);

}

}

public class TestHipopotamos {

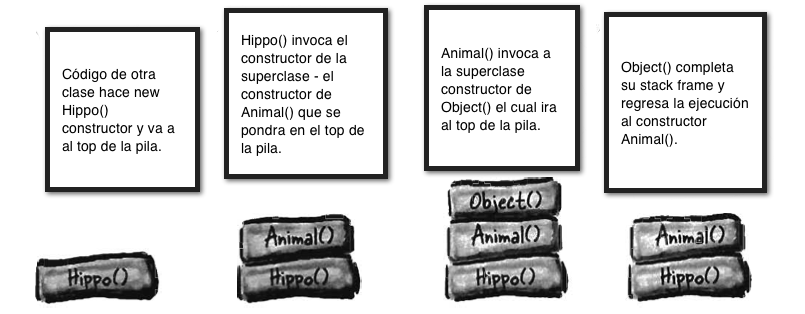
public static void main (String[] args) {

System.out.println(“Iniciar ….”);

Hipopotamo h = new Hipopotamo();

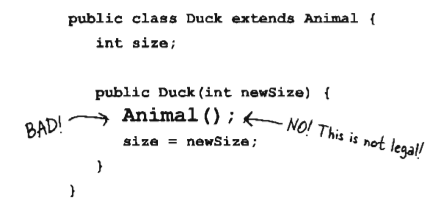
}

}

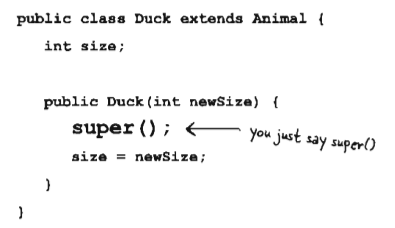


## 2.5 Como invocas al constructor de una super clase

No hagas lo siguiente:

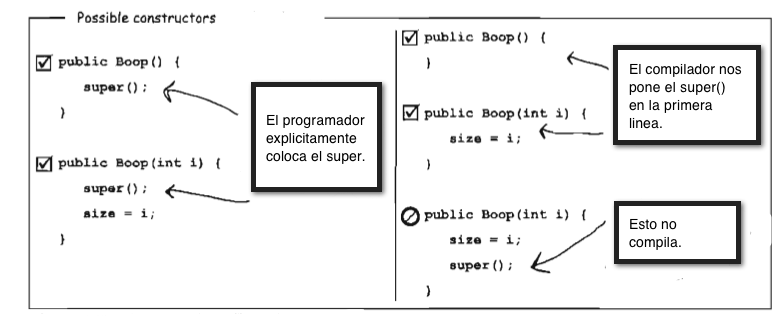


En su defecto debes usar **super():**



Una llamada a super() en tu constructor pone a la superclase constructor en el top de la pila. Y ese superclass constructor llama a su superclase constructor. Y as hasta que Object constructor esta en el top de la pila. Una vez que Object finaliza, es retirado de la pila y pasa al siguiente.

**OJO: La llamada a Súper debe ser antes de cualquier sentencia en los constructores**



## 2.6 Constructores de Superclases con argumentos

Veamos como cuando tenemos constructores con argumentos podemos usar super

