**Academia Java – JAVA**

***Vivir en la ciudad de objetos***

Tabla de contenido

1. Recordando el ejemplo de las sillas 3

1.1 Qué hay acerca del método rotate() de Ameba 4

1.3 Entendiendo Herencia 4

1.4 Ejemplo de Herencia 6

1.5 Vamos a disenar el arbol de herencia para un programa de simulacion animal 8

1.6 ¿Qué metodo es invocado? 13

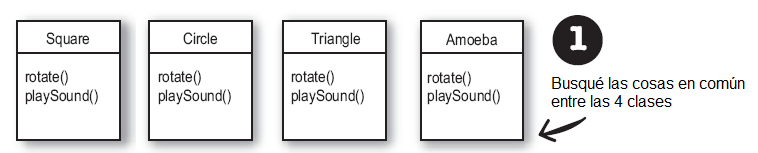
1.7 Usando IS-A y HAS-A 15

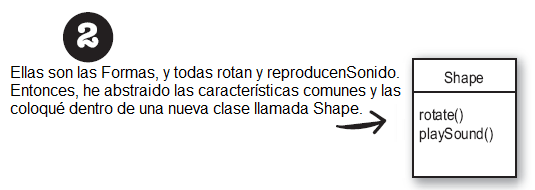
1.8 Cuando disennas usando herencia estas USANDO o estas ABUSANDO? 16

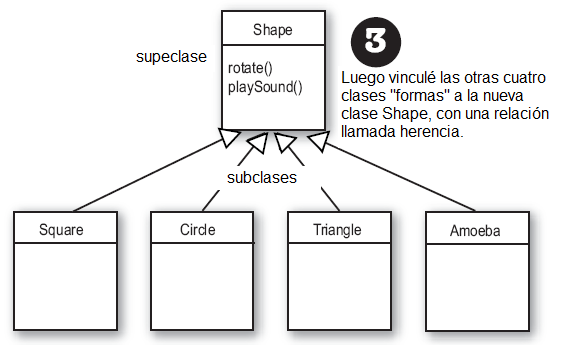
# 

# Recordando el ejemplo de las sillas

Recordemos la guerra de sillas entre Larry (el programador procedural) y Brad (programador orientado a objetos)







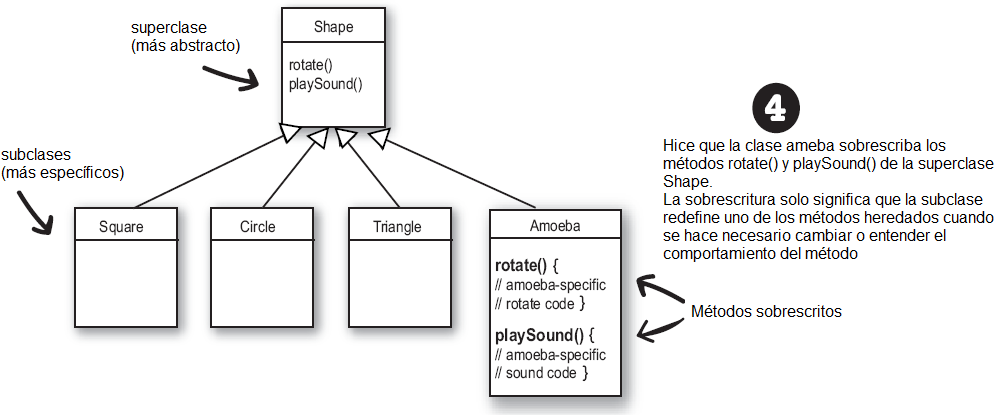
## 1.1 Qué hay acerca del método rotate() de Ameba

Larry: Bueno, ese no es el gran problema aquí – sino que la forma de Ameba tiene los procedimientos rotate() y playSound() diferentes?

Brad: Métodos.

Larry: Como sea. ¿Cómo la ameba puede hacer algo diferente si “hereda” la funcionalidad de la clase Shape?

Brad: Ese es el último paso. La clase Ameba sobrescribe los métodos de la clase Shape. Luego, en tiempo de ejecución, la JVM sabe exactamente que método rotate() ejecutar cuando alguien le pida a Ameba girar.



## 1.3 Entendiendo Herencia

# 

Cuando disenas con herencia, colocas el codigo comun en una clase y luego le dices a las clases mas especificas que la clase comun (mas abstracta) es su superclase. Cuando una clase hereda de otra, la **subclase hereda de otra superclase.**

En Java, nosotros decimos que la **subclase hereda de la superclase.** Una relacion de herencia significa que la subclase hereda los **miembros** de una superclase. Cuando nosotros decimos los “miembros de una clase” eso hace referencia a las variables de instancia y metodos.

Por ejemplo, si Spiderman es una subclase de SuperHeroe. La clase Spiderman AUTOMATICAMENTE hereda las variables de instancia y metodos comunes de los superheroes incluyendo vestimenta, poderesEspeciales, usarPoderesEspeciales(), entre otros. Pero, la clase Spiderman **puede anadir nuevos metodos y variables de instancia** de su propiedad, y de esta forma **sobreescribir los metodos heredados de la superclase** SuperHeroe.

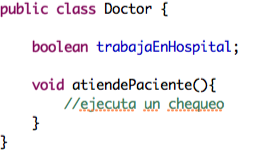
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  | | --- | | SuperHeroe (SuperClase) | | vestimenta  poderesEspeciales | | ponerseVestimenta()  quitarseVestimenta()  usarPoderesEspeciales() |  |  | | --- | | Spiderman (Subclase) | |  | | usarPoderesEspeciales()  ponerseVestimenta() | |

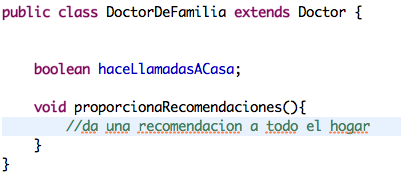
Spiderman necesita requerimientos especiales para sus poderes especiales y vestimenta. Por esa razon, sobreescribe dichos metodos.

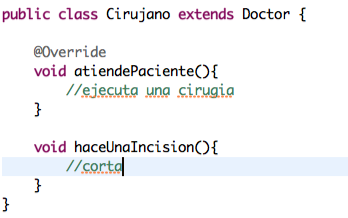
Tal vez otro heroe no necesite sobreescribir nada o sobreescribir algunos, como?

|  |  |
| --- | --- |
|  |  |

## 1.4 Ejemplo de Herencia







|  |  |
| --- | --- |
| C:\Users\ingas\Desktop\Atencion2.png | En el ejemplo anterior:   1. Cuantas variables de instancia tiene Cirujano? \_\_\_ 2. Cuantas variables de instancia tiene DoctorDeFamilia? \_\_\_ 3. Cuantos metodos tiene Doctor? \_\_\_ 4. Cuantos metodos tiene Cirujano? \_\_\_ 5. Cuantos metodos tiene DoctorDeFamilia? \_\_\_ 6. Puede un DoctorDeFamilia usar atiendePaciente()? \_\_\_ 7. Puede un DoctorDeFamilia usar haceUnaIncision()? \_\_\_ |

Otro ejemplo de Doctor:



## 1.5 Vamos a disenar el arbol de herencia para un programa de simulacion animal

Imagina que se te ha solicitado el diseño de un programa de simulación que permita a los usuarios ver a diferentes animales en su entorno y ver que sucede. Vamos a concentrarnos entonces en el diseño.

Nos han dado una lista de algunos animales que estarán en el programa, pero, no todos. Nosotros sabemos que cada animal estará representado por un objeto, y que dichos objetos se moverán en su entorno, haciendo lo que necesita cada uno de acuerdo a su tipo particular.

**Nosotros deseamos que otros programadores estén facultados para añadir nuevos tipos de animales al programa en cualquier momento.**

Primero tenemos que concebir las características mas comunes, abstractas que todos los animales tienen, y construir con dichas características una clase de la cual todos los animales puedan heredar.

1. Hay que buscar objetos que tengan atributos comunes y comportamientos

Que hacen estos seis tipos en común? Esto ayuda a abstraer sus comportamientos (paso 2)

Como estos tipos se relacionan? Esto ayuda a definir las relaciones de herencia? (paso 4-5)

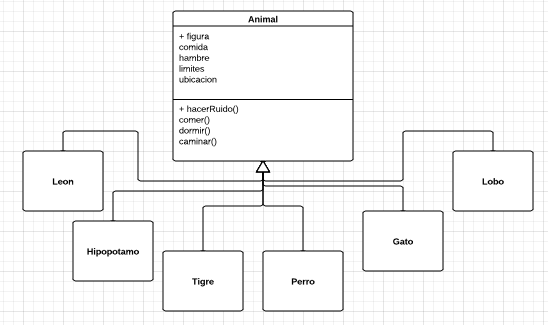
|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

1. Usando herencia para evitar codigo duplicado en cada subclase.

El objetivo es diseñar una clase que represente el estado y comportamiento común.

**Estos objetos son todos animales, así que haremos una clase común o superclase llamada Animal.**

**Pondremos todos los métodos y variables de instancia que todos los animales podrían tener.**



Tenemos 5 variables de instancia:

**figura** – el archivo que representa la imagen JPEG del animal

**comida** – el tipo de comida que el animal come. Ahí puede haber solo dos valores: carne o vegetal

**hambre** – un entero representando el nivel de hambre del animal. Este valor dependerá de cuanto el animal come.

**limites** – valores que representan la altura y ancho del espacio (ejemplo: 640x480) que los animales necesitan para vivir.

**ubicación** – las coordenadas X e Y donde el animal estará en el espacio asignado.

Tendremos 4 métodos:

**hacerRuido**() – para que el animal haga ruido

**comer**() – para que el animal coma carne o vegetal

**dormir**() – para que el animal duerma

**caminar**() – mientras no duerma o coma

1. Todos los animales comen lo mismo o de la misma manera?

**Hay que decidir si una clase necesita comportamiento (implementación de métodos) que son específicos a un tipo de subclase en particular.**

**Vemos en la clase Animal y decidimos que comer() y hacerRuido() será sobrescrito por las subclases individuales.**

|  |  |  |
| --- | --- | --- |
|  |  |  |

1. Hay que buscar mas oportunidades de herencia.

**Oportunidades para usar abstracción, encontrar dos o mas subclases que puedan necesitar comportamiento común.**

**Buscar nuestras clases y ver que Lobo y Perro pueden tener comportamiento común y lo mismo sucede con León, Tigre y Gato.**

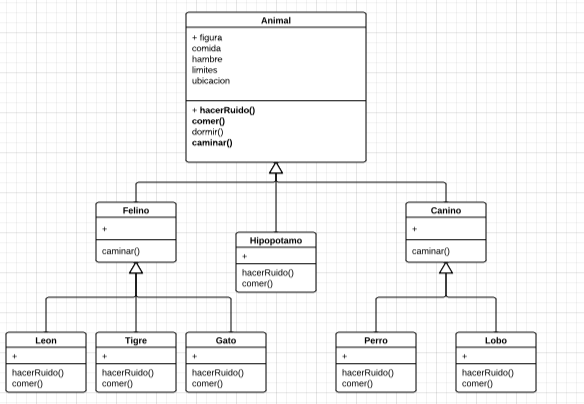
****

**umm! Lobo y Perro son ambos caninos.**

**León, Tigre y Gato son felinos.**

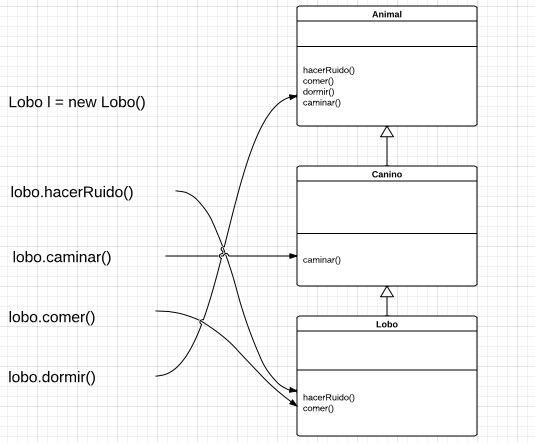
1. Finalizando la jerarquia de clases.

Puesto que ya hemos organizado los animales. Haremos un nivel que potencie nuestro diseño de clases. Decidimos que los caninos usaran un método común denominado caminar(), porque ellos tienden a moverse en grupos. Mientras que los felinos tienden a evitar caminar con otros de su propio tipo, así que definitivamente su método caminar() es diferente. Con el Hipopótamo continuaremos usando el método caminar() genérico de Animal.



## 1.6 ¿Qué metodo es invocado?

La clase Lobo tiene cuatro métodos. Uno hereda de Animal, otro de Canino y dos sobrescritos de la clase Lobo. Cuando tu creas un objeto Lobo y asignas a este una variable, tu puedes usar el operador “.” En esa variable de referencia para invocar los cuatro métodos. Que versión de estos métodos es llamado?



Cuando tu llamas al metodo de una referencia de objeto, tu estas llamando a la version mas especfica del metodo para ese tipo de objeto.

En otras palabras, **el mas cercano gana!.**

El mas cercano en el arbol de la herencia. Canino es mas cercano que Animal y Lobo es mas cercano que canino. Asi que invocar un metodo en una referencia de un objeto Lobo le dice a la JVM que comience a buscar en primer lugar desde la clase Lobo. Si el JVM no encuentra una version de ese metodo en la clase Lobo, comienza a buscar en la jerarquia de clases hasta que encuentra uno que corresponda.

## 1.7 Usando IS-A y HAS-A

Recuerda que cuando una clase hereda de otra, nosotros decimos que la subclase hereda de la superclase. Cuando tu deseas saber si alguna cosa debería heredar de otra, aplica el test **IS-A**.

Un triangulo IS-A (es una) Figura.

Gato IS-A Felino

Cirujano IS-A Doctor

Una bannera hereda de Banno. Suena razonable hasta que le aplicas el test IS-A.



Para saber si estamos disennando correctamente nuestros tipos, nos preguntamos, “tiene sentido decir que el tipo X es un tipo de Y?” Si no lo tiene, entonces sabemos que algo malo esta pasando con el disenno, asi que si aplicamos el test IS-A con Bannera IS-A Banno. Definitivamente es Falso.

Lo inverso tampoco funciona. Un Banno IS-A Bannera. Tampoco trabaja.

Bannera y Banno estan relacionados, pero, no, a traves, de herencia. Una Bannera y un Banno estan ligados por una relacion HAS-A (contiene un). Ahora si tiene sentido decir “Un Banno HAS-A Bannera”.

Asi es, un Banno tiene una Bannera. En otras palabras la clase Banno tiene una reencia a una Bannera, pero el Banno no extiende de Bannera y viceversa.

|  |  |  |
| --- | --- | --- |
|  |  |  |

Un Banno HAS-A Bannera y una Bannera HAS-A Burbujas.

Pero nadie hereda o extiende de ninguna.

|  |  |
| --- | --- |
| C:\Users\ingas\Desktop\Atencion2.png | Existen cuatro niveles de acceso. Desde el **mas restrictivo** al **menos restrictivo**. Los cuatros niveles son:  **PRIVATE** **DEFAULT** **PROTECTED** **PUBLIC**  **Los miembros publicos son heredados**  **Los miembros privados no son heredados** |

## 1.8 Cuando disennas usando herencia estas USANDO o estas ABUSANDO?

Sigue estas reglas para apoyarte a crear buenos diseños de herencia.

**USA** la herencia cuando encuentras una clase mas especifica que el tipo de una superclase. Ejemplo: Un Sauce es un tipo especifico de Árbol. Así el Sauce heredara de Árbol y tiene mas sentido.

**USA** la herencia cuando tengamos comportamiento (código implementado) que debe y puede ser compartido entre múltiples clases de un mismo tipo general. Ejemplo: circulo, cuadrado, rectángulo, triangulo necesitan rotar y tocar sonido, así que ponemos dicha funcionalidad en la superclase Figura. Esto tiene mas sentido y hace mas fácil el mantenimiento y escalabilidad. En general la herencia es buena forma de conseguir comportamiento a reutilizar pero no es necesariamente la mejor elección. Existen patrones de diseño que nos ayudaran a ver mas opciones flexibles.

Otro tema muy importante que debes anotar es **Necesito saber patrones de Diseño**.

**NO USAR** herencia si la relación entre una clase y subclase viola las anteriores dos reglas. Por ejemplo, imagina tu escribir un código especial de impresión para la clase Alarma y ahora que tu necesitas código de impresión en la clase Piano, tu quieres que Piano herede de Alarma de manera que herede el código de impresión. Eso no tiene sentido! Un Piano no es un tipo especifico de Alarma. (Así que el código de impresión debería ser una clase nueva Impresión, y las demás clases tendrán una relación HAS-A con ella para usar sus métodos).

|  |  |
| --- | --- |
| * Una subclase hereda de una superclase * Una subclase hereda todos los metodos publicos y variables de instancia de la superclase, pero no hereda, las variables y metodos privados de la superclase. * Metodos heredados pueden ser sobreescritos; variables de instancia no pueden ser sobreescritos (aunque pueden ser redefinidos en la subclase, pero, eso no es la misma cosa y casi nunca necesitan hacerlo). * Usa el test IS-A para verificar que tu jerarquia de herencia es valida. Si X hereda de Y, entonces X IS-A Y debe tener sentido. * La relacion IS-A trabaja en solo una direccion. Un Hipopotamos es un Animal, pero, no todos los animales son Hipopotamos * Cuando un metodo es sobrescrito en una subclase, y ese metodo es invoado en una instancia de la subclase, la veresion sobrescrita del metodo es invodado (el mas cercano gana). * Si la clase B hereda de A, y C hereda de B, la clase B IS-A clase A, y clase C IS-A clase B y clase C tambien IS-a clase A. | C:\Users\ingas\Desktop\Atencion2.png |

# Polimorfismo

**Para ver como trabaja, vamos a revisar como normalmente declaramos una referencia y creamos un objeto.**

|  |  |
| --- | --- |
|  |  |
| **Dog myDog** = **new** Dog();  Le dice a la JVM que reserve un espacio para una variable de referencia, y la nombra myDog. La variable de referencia será siempre de tipo Dog. En otras palabras, control remoto que tiene botones para controlar a un Dog, no a un Cat o un Button o un Socket. |
|  |  |
| Dog myDog = **new Dog();**  Le dice a la JVM que reserve un espacio para el objeto new Object en el heap. |
|  |  |
| Dog myDog **=** new Dog();  Asigna new Dog a la variable de referencia myDog. |

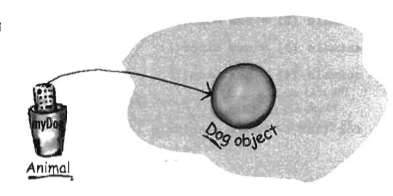
**El punto importante aquí es que el tipo de referencia Y el tipo de objeto son el mismo.**

**En este ejemplo, ambos son Dog.**



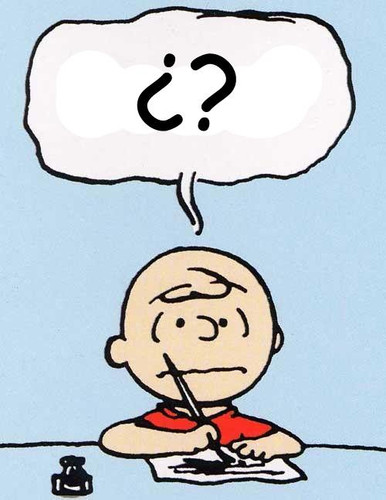
**Pero con polimorfismo la referencia y el objeto pueden ser diferentes.**

Animal myDog **=** new Dog();

****

**Con polimorfismo el tipo de referencia puede ser una superclase del tipo de objeto actual**

Cuando tu declaras una variable de referencia, cualquier objeto que pasa el test IS-A para el tipo declarado de la variable de referencia puede ser asignado a esa referencia. En otras palabras, cualquier cosa que extienda del tipo de variable de referencia declarada puede ser asignado a la variable de referencia. **Esto nos permite cosas como los arrays polimórficos.**

****

**Vamos a hacer un ejemplo que ayude a entender esto**

Animal [] animales = new Animal[5];

animales[0] = new Perro();

animales[1] = new Gato();

animales[2] = new Lobo();

animales[3] = new Hipopotamo();

animales[4] = new Leon();

for (int i=0; i < animales.length; i++) {

animales[i] .comer();

animales[i].caminar();

}

**Hay mas que esto!.**

**Podemos tener argumentos y tipos de retorno polimórficos.**

Si hemos podido declarar una variable de referencia de un supertipo como Animal y asignamos a este un objeto de una subclase de este, digamos Perro; como trabajaría esto cuando la referencia es un argumento de un método.

**Lobo** o **Hipopotamo** o **Perro** o **Gato**

class Veterinaria {

public void tratarAnimal(Animal a) {

a.hacerRuido();

}

}

class Owner {

public void iniciar(){

Que método se ejecuta?

Veterinaria v = new Veterinaria();

Perro p = new Perro();

Hipopotamo h = new Hipopotamo();

v.tratarAnimal(p);

v.tratarAnimal(h);

}

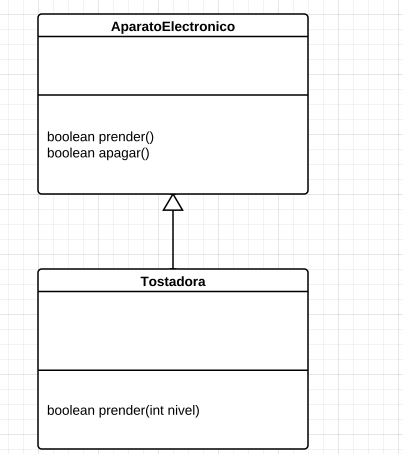
}

**CONCLUSION: Con el polimorfismo, podemos escribir código que no necesita cambiar cuando tu introduces nuevos subtipos en tu programa.**

# Mantener un contrato: reglas para sobrescritura

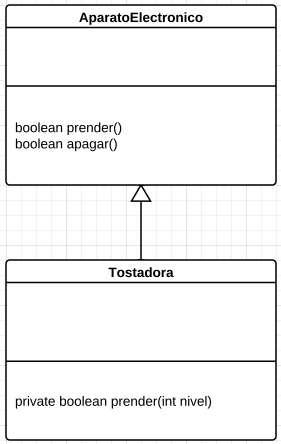
Cuanto tu sobrescribes un método de una superclase tu estas de acuerdo con el contrato. El contrato dice, por ejemplo, “no tomare argumentos y retorno un boolean”. En otras palabras, los argumentos y tipos de retornos del método que tu sobrescribes deben ser exactamente igual al método sobrescrito de la superclase.

**Los métodos son el contrato.**



En la figura siguiente no tenemos una sobre escritura. **No se puede cambiar los argumentos en un método sobrescrito**.

**Tampoco se puede cambiar el nivel de acceso.**

****

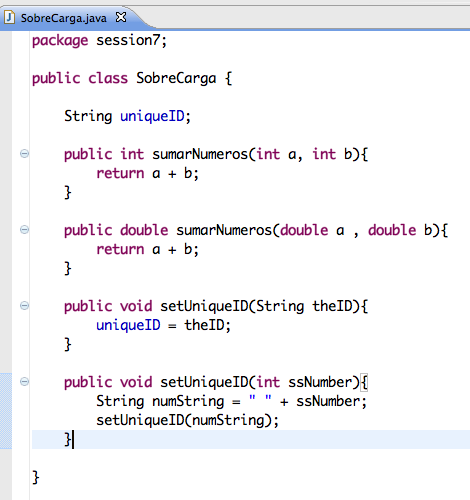
**Si se ve en el ejemplo se ha cambiado a private el nivel de acceso.**

# Sobrecarga de un método

**Una sobrecarga de un método es solo un método diferente que tiene el mismo nombre de un método de su superclase. Pero no es resultado de herencia. No es lo mismo que un método sobrescrito.**

* **Los tipos de retorno pueden ser diferentes**
* **No puedes cambiar SOLO el tipo de retorno.** El compilador te pedira que cambies los argumentos tambien.
* **Tu puedes variar los niveles de acceso en cualquier direccion.** Puedes hacerlo mas restrictivo.

Ejemplos legales de overloading (sobrecarga):

****