**Academia Java – JAVA**

***Polimorfismo Serio***

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)

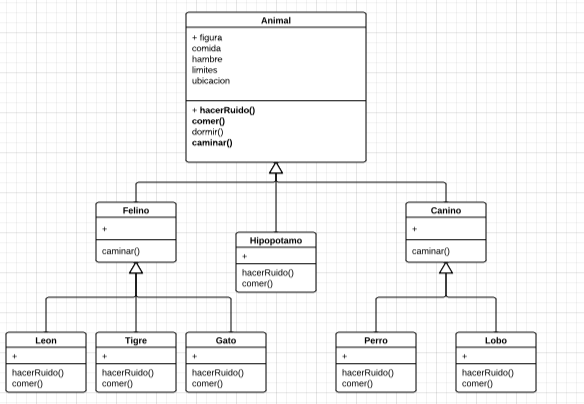
[1.6 Que pasa si necesitamos cambiar el contrato? 17](#_Toc313542555)

[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)

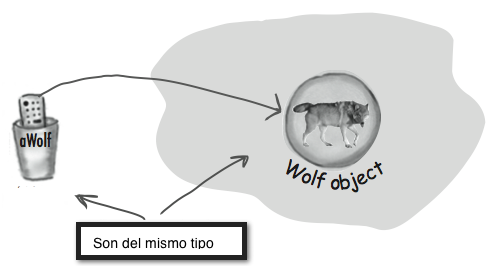
# No olvides esto cuando diseñas una jerarquía como esta



La estructura de clases no esta mal. Con un diseño así evitamos duplicar código a su mínima expresión, y sobrescribimos los métodos que pensamos deben tener una implementación especifica. Gracias al polimorfismo podemos trabajar con cualquier animal incluso con animales que no se han pensado aun para el proyecto. Hemos ya definido el protocolo o contrato común que deben seguir los animales (cuatro métodos que todos los animales tienen) en la superclase Animal, y estamos listos para comenzar a hacer nuevos Leones y Tigres o Hipopótamos.

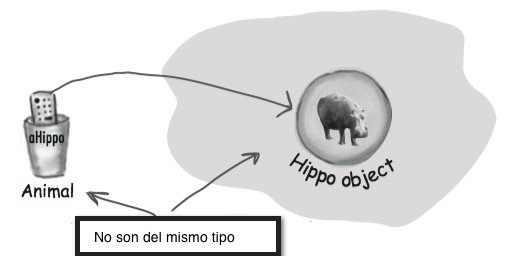
**Nosotros podemos decir:**

Lobo aWolf = new Lobo();



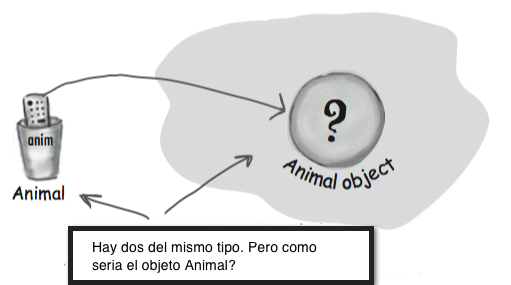
**Y también podemos decir:**

Animal aHippo = new Hipopotamo();



**Y que sucede con esto:**

Animal anim = new Animal();



## 1.1 Como es un objeto resultado de new Animal()?

**Como son los valores de estas variables de instancia??**

**Algunas clases no deberian ser instanciadas!**

Esto tiene sentido porque si pensamos en un objeto Lobo, Hipopotamo, Tigre, sabemos que valores colocar en variables de instancia. Pero, un animal?

Ahora el problema, es que si necesitamos una clase Animal para herencia y polimorfismo, pero, solamente se instancie subclases de Animal, pero, no Animal en si.

Nosotros deseamos objetos Tigre y Leon, pero no objetos Animal.

Afortunadamente, hay una forma de prevenir que se instancie a partir de una clase. Para eso hay que marcar la clase con **abstract,** de manera que el compilador evite cualquier codigo que intente crear una instancia de ese tipo.

Cuando tu diseñas tus clases. Tienes que decir que clases serán **abstractas** y cuales seran **concretas**. Las clases concretas son aquellas de las cuales podemos crear instancias. Una clase concreta significa que esta HABILITADA para crear objetos.

***abstract class Canino extends Animal {***

***public void caminar() { }***

***}***

## 1.3 El compilador no te permitirá instanciar una clase abstracta

# 

Una clase abstracta significa que nadie puede hacer una nueva instancia de esa clase. Nosotros podemos usarla como un tipo de referencia, para el polimorfismo, pero, no podemos hacer objetos de ese tipo. El compilador nos garantiza esto.

***abstract class Canino extends Animal {***

***public void caminar() { }***

***}***

public class HacerCaninos {

public void go() {

Esto es OK.

Canino c;

c = new Perro();

c = new Canino();

c.caminar();

La clase Canino es abstract. Tendremos un error de compilación.

}

}

Una clase abstracta no tiene uso virtual, ni valor, ni proposito en la vida, a no ser que sea **extended** (heredado).

Con una clase abstracta, los que hacen el trabajo en tiempo de ejecucion son **instancias** **de una subclase** de tu clase abstracta.

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

## 1.4 Abstracta vs Concreta

Una clase que no es abstract es llamada Concreta. En el arbol de herencia Animal, si hacemos abstracto a Animal, Canino y Felino, las demas subclases seran concretas.

Si buscas en las librerias Java (Java API), encontraremos muchas clases abstractas sobretodo en la librería GUI (Grafic User Interface). Por ejemplo un componente GUI es la superclase de clases como Buttons, Text Areas, Scrollbars, dialog boxes, etc. Nosotros no tenemos porque hacer una instancia de Component y ponerlo en la pantalla, nosotros en cambio hacemos un JButton. En otras palabras tu instancias solo una subclase concreta de Component, pero nunca el Component en si.

**Metodos Abstractos**

Asi como las clases, tu puedes tener metodos abstractos tambien. Una clase abstracta significa que la clase debe ser heredada; un metodo abstracto significa que el metodo debe ser sobreescrito. Nosotros podemos decidir que algunos (o todos) los comportamientos en una clase abstracta no tengan ningun sentido a no ser que ellos sean implementados por una subclase mas especifica. En otras palabras, tu no puedes pensar en la implementacion de un metodo generico que satisfaga a todas las subclases. Por ejemplo, el metodo comer() como seria para que satisfaga a todos?



**Un metodo abstracto no tiene cuerpo**

Como hemos decidido que no hay codigo que tenga sentido en el metodo abstracto, no podremos nada en el cuerpo del metodo. Asi que no hay llaves – solo la declaracion y un ;

**public abstract void comer();**

**Si tu declaras un método abstract, tienes que marcar la clase como abstract también. No puedes tener un método abstract en una clase no abstract.**

Si pones un simple metodo abstract en una clase, tienes que hacer la clase abstract. Pero, podemos hacer una mixtura entre metodos abtractos y concretos en una clase abstracta.

**Tenemos que implementar todos los metodos ABSTRACT**



Los métodos abstract no tienen cuerpo; ellos solo sirven para el polimorfismo. Esto significa que la primera clase concreta en el árbol de herencia debe implementar todos los métodos abstract. En el caso de nuestra jerarquia de animales, Perro tiene que implementar los metodos abstract de Animal y Canino.

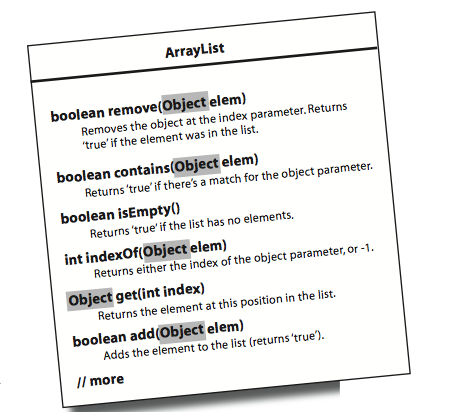
Ahora recordemos que una clase abstracta puede tener ambos metodos abstractos y concretos. Si Canino implementa un metodo abstracto de Animal, Perro ya no tendra que hacerlo.

Cuando decimos que debe implementar un metodo abstracto, esto significa que debe proveer un cuerpo (body). Esto significa que debes crear un metodo concreto en tu clase con la misma firma (nombre y argumentos) y un tipo de retorno que es compatible con el tipo de retorno declarado del metodo abstracto.

## 1.5 Cada clase en Java hereda de Object

La clase Object es la madre de todas las clases; esto es la superclase de todo.

Recuerda la clase ArrayList, esta clase esta pensada para trabajar con cualquier tipo. Entonces se declara sus argumentos y tipos de retorno de tipo Object para poder trabajar con cualquier subtipo de el.



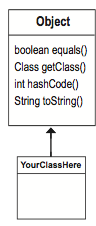
Recuerda, **Cualquier clase que explicitamente no hereda de ninguna clase, implicitamente hereda de Object.**

En nuestro ejemplo de arbol de animales. Perro hereda de Canino, asi que no hereda directamente de Object (aunque lo hace indirectamente), y lo mismo para Canino, aunque Animal si hereda directamente de Object.

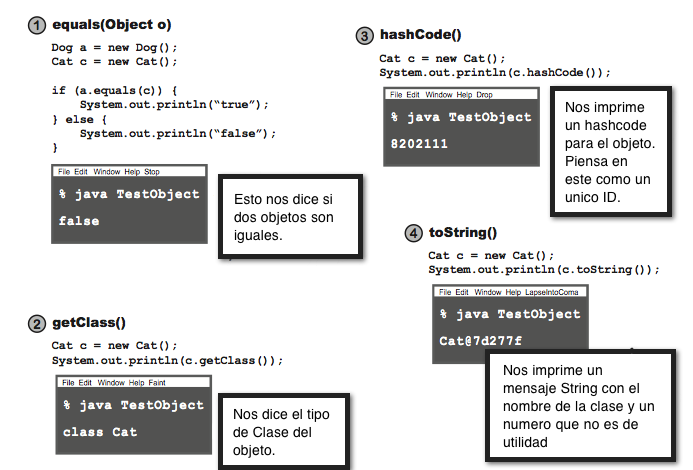
**Que hay en esta ultra-super-megaclase Object?**

Si fueramos Java, que comportamiento deberia tener cada objecto? Veamos que hay acerca de tener un metodo que nos permita encontrar si un objeto es igual a otro objeto? Que hay de otro metodo que pueda decirnos de que tipo de clase es ese objeto? Tal vez necesitemos de un metodo que nos de un hashcode para ese objeto de modo que podamos usar ese objeto en hashtables. Y finalmente un buen metodo seria aquel que imprima un mensaje String para ese objeto.

Y bueno esto descrito anteriormente es lo que tiene la clase Object. Cuatro metodos que hacen esas cuatro cosas. Esos no son todos, pero, al menos de estos cuatro hay que tener mucha atencion:



Estos son algunos de los metodos de la clase Object.

****

**Usar referencias polimorficas de tipo Object tiene un precio…**

Antes de querer usar Object para todos tus ultra-flexible argumentos y tipos de retorno, ten en cuenta esta seccion al usar el tipo Object como una referencia.

Cuando ponemos un objeto en un ArrayList<Perro>, este sera un Perro y retornara como Perro:

ArrayList**<Perro>** miArrayListPerros = new ArrayList**<Perro>**();

Perro p = new Perro();

miArrayListPerros.add(p);

Perro d = miArrayListPerros.get(0); // esto funciona sin problemas

Pero, que sucede si declaramos este como un ArrayList<Object>:

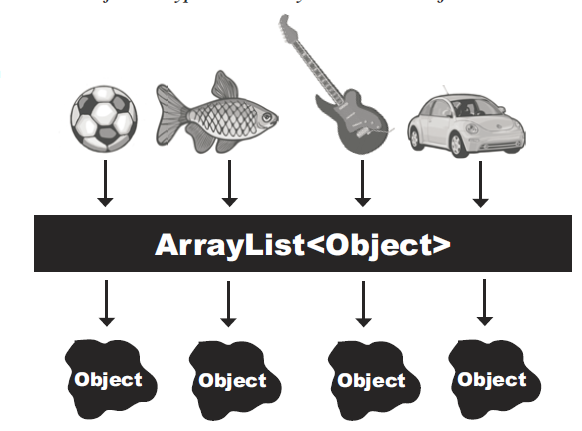
ArrayList**<Object>** miArrayListPerros = new ArrayList**<Object>**();

Perro p = new Perro();

miArrayListPerros.add(p);

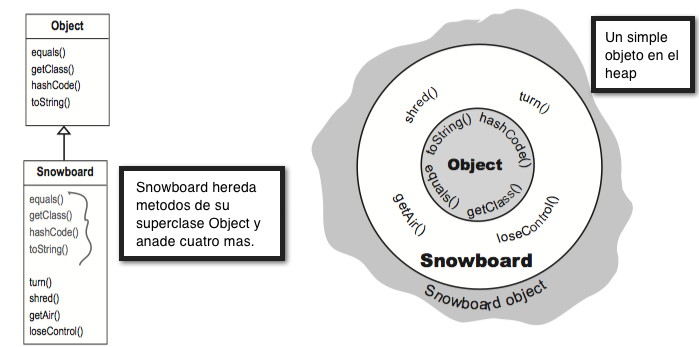
~~Perro d = miArrayListPerros.get(0);~~ // **ESTO YA NO FUNCIONA**

Todo lo que se obtenga de un ArrayList<Object> es de tipo Object, sin importar si lo que annadiste a la lista era de otro tipo.

****

**Veamos internamente un objeto**

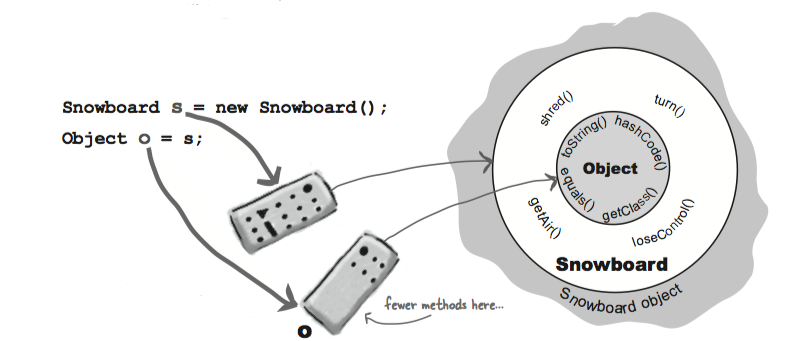
Un objeto contiene todo lo que hereda de cada uno de sus superclases. Esto significa que cada objeto asi como es del tipo actual de su clase, tambien es una instancia de la clase Object. Esto significa que cualquier objeto en java puede ser tratado no solo como un Perro, Boton, Snowboard, sino tambien como un objeto. Cuando tu dices new Snowboard(), tu estas obteniendo un simple objeto en el heap – un objeto Snowboard – pero, que tiene tambien una representacion interna de la clase Object en si mismo.



Hay un solo objeto en el heap. Un objeto Snowboard que contiene partes de la clase SnowBoard y de la clase Object en si mismo.

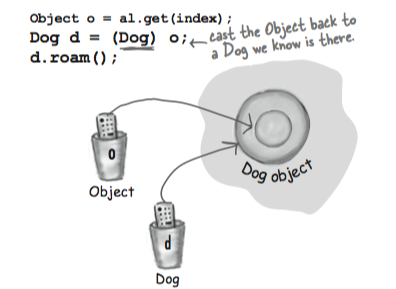
**Polimorfismo significa ‘muchas formas’**

**Podemos tratar a Snowboard como un Snowboard o como un Objeto.**



**Casting a una referencia Object para que regrese a su tipo real**

El casting funciona si es que tenemos la seguridad de que el objeto es realmente un Perro.



Si no estas seguro que es un Perro, tenemos que usar el operador **instanceof** para chequear eso. Esto es importante, porque si nos equivocamos al hacer el cast obtendremos un ClassCastException en tiempo de ejecucion y no esperamos eso.

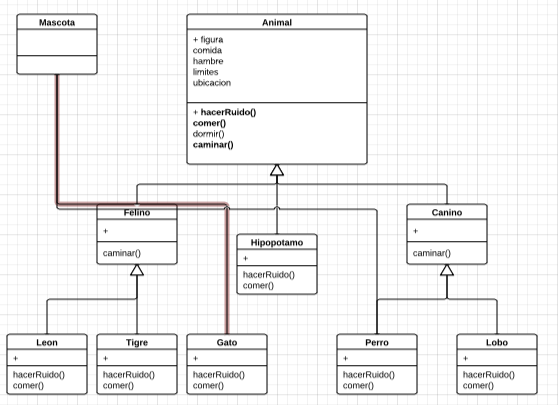


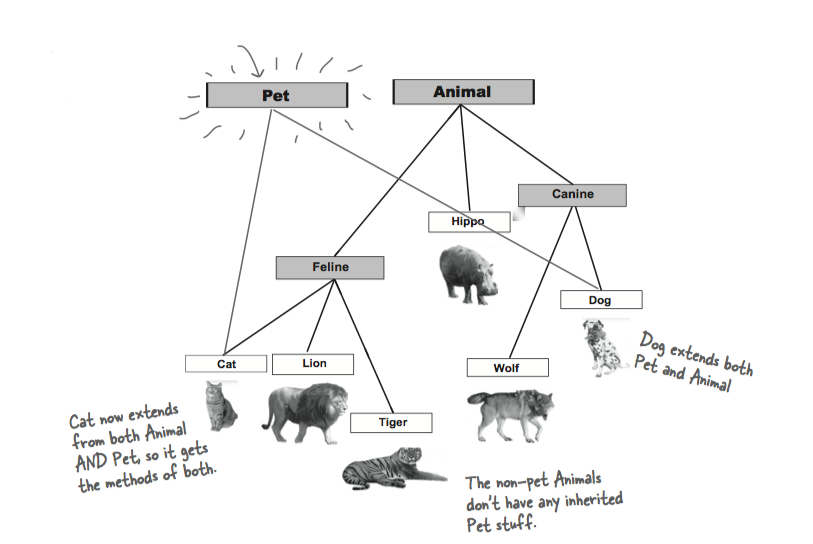
**Todo esto que hemos visto ha sido importante para ver que solo podemos llamar a un metodo de un objeto, si y solamente si, la variable de referencia tiene ese metodo.**

## 1.6 Que pasa si necesitamos cambiar el contrato?

Nos han solicitado que nuestras clases se puedan utilizar para un programa de mascotas PetShop.

De esta manera nos piden que ciertas subclases tengan comportamiento de Mascota.





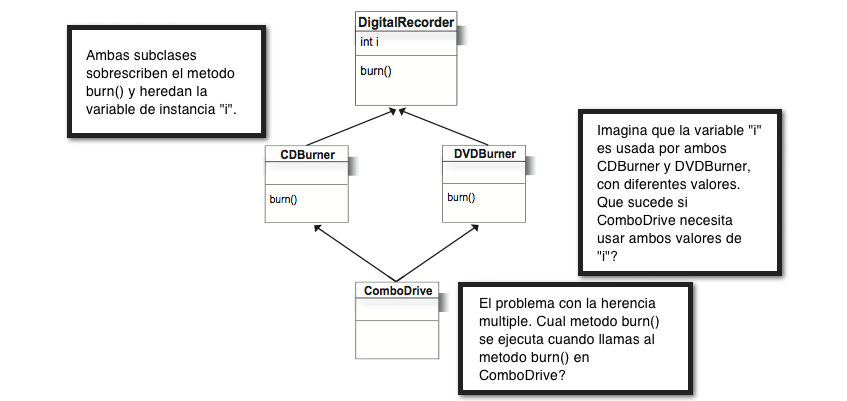
**Necesitariamos dos superclases en el top.**

Haremos una nueva superclase abstracta llamada Mascota que tenga los metodos de una mascota.

**Lo que acabamos de ver es denominado “Herencia Multiple” y puede realmente ser una cosa muy mala.**

Claro, si fuese posible hacerlo en Java. Pero, esto no es posible, porque la multiple herencia tiene un problema conocido como el DDD (Deadly Diamond of Death – Diamante mortal de la muerte)

**El Deadly Diamond of Death**



Un lenguaje que soporte el DDD annade muchas complejidades, porque tiene que tener reglas especiales para lidiar con las potenciales ambiguedades. Y reglas extras significa trabajo extra para aprender, ver, y usar estos “casos especiales”. Java ha sido desarrollado para ser simple, con reglas consistentes que no tiene doble escenerio. Asi que Java (a diferencia de C++) nos protege de tener que pensar en el DDD.

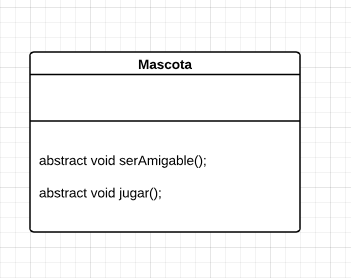
Pero, igual esto no nos aleja del problema principal. *Como hacemos que una cosa sea Animal/Mascota según nuestros requerimientos*?

## 1.7 Interfaces al Rescate

Java nos da una solucion. Una interface. No una interface de usuario (GUI), sino el uso de la palabra reservada **interface.**

Una interface soluciona el problema de la herencia multiple dandonos mucho de los beneficios de la herencia multiple sin el dolor de lidiar con el Dealy Diamond of Death (DDD).

La forma como trabajan las interfaces es simple: **es una colección de metodos abstractos**. Esto es la subclase **debe** implementar los metodos, de manera que en tiempo de ejecucion el JVM no se confunda acerca de cual de las dos versiones heredadas debe llamar.



**Una interface Java es una clase abstracta pura al 100%**

**Para DEFINIR una interface:**

public interface Mascota {…}

**Para IMPLEMENTAR una interface:**

public class Perro extends Canino implements Mascota { …}

Cuando usas implements, también estas heredando de esa clase.

**Ejercicio: Hacer e Implementar una interface Mascota**

**public interface Mascota {**

**public abstract void serAmigable();**

**public abstract void jugar();**

**}**

public class Perro extends Canino implements Mascota {

public void serAmigable() {

//do

}

public void jugar() {

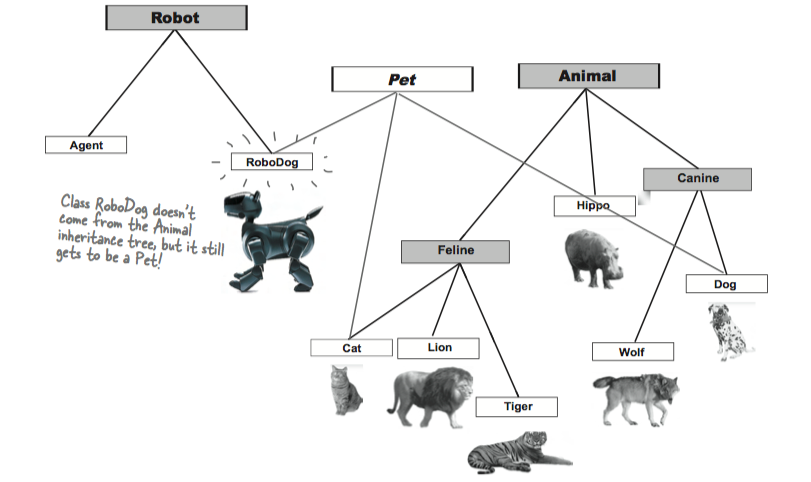
//do

}

//otros metodos

}

Y de esta forma… **Clases de diferentes arboles de herencia pueden implementar la misma interface.**



**Esto es lo mejor, una clase puede implementar multiples interfaces.**

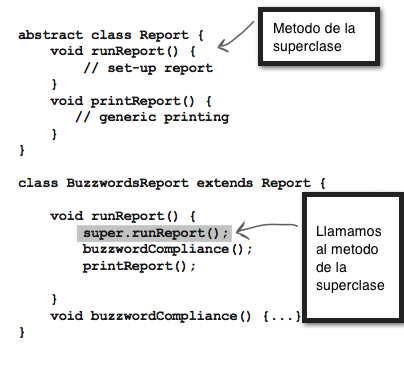
public class Dog extends Animal implements Mascota, Grabable, Dibujable { ….}

## 1.8 Como sabes cuando hacer una clase, subclase, una clase abstracta o una interface

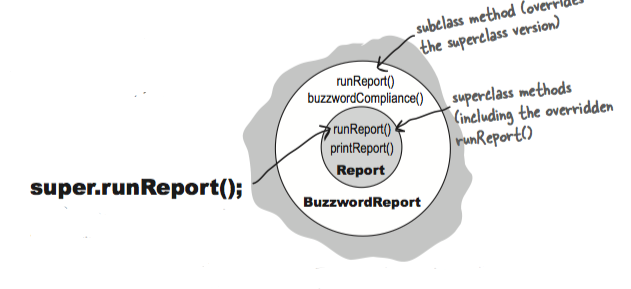
* Construye una clase que no hereda de nada (solo Object) cuando tu nueva clase no pasa el test IS-A para otro tipo que tengas.
* Construye una subclase (hereda de otra clase) cuando tu necesites hacer una version **mas especifica** de una clase que necesitas sobreescribir o para annadir nuevo comportamiento.
* Usa una clase abstracta cuando tu deseas definir un **template** para un grupo de subclases, y tu tiene al menos un codigo de implementacion que todas las subclases pueden usar. Convierte la clase a abstract cuando te deseas asegurar que nadie haga instancias u objetos a paritr de ese tipo.
* Usa una interface cuando deseas definir un **rol** que otras clases puedan realizar, sin importar si pertenecen al mismo arbol de herencia.

## 1.9 Usando super

Cuando querramos invocar el metodo de una superclase, usamos super.



El super te permite ingresar a la parte interior heredada e invocar dicho metodo.



|  |  |
| --- | --- |
| * Cuando tu no deseas que una clase sea instanciada (en otras palabras que nadie haga un nuevo objeto de ese tipo) marcamos la clase con abstract. * Una clase abstract puede tener metodos abstractos y no abstractos * Si una clase tiene un metodo abstracto, la clase debe ser marcada como abstracta * Un metodo abstracto no tiene cuerpo, y la declaracion termina con “;” ( no hay llaves) * Todos los metodos abstractos deben ser implementados en la primera subclase concreta en el árbol de herencia * Cada clase en Java es indirecta o directa subclase de Object (java.lang.Object) * Los metodos pueden ser declarados con argumentos Object y/o tipos de retorno * Tu puedes llamar metodos de un objeto si y solamente si los metodos estan en la clase (o interface) usado como tipo de variable de referencia, sin importar el tipo actual del ojecto. Asi una variable de referencia de tipo Object puede ser usado solamente para llamar a metodos definidos en la clase Object, en lugar del tipo del objeto al cual la referencia apunta. * Una variable de referencia de tipo Object puede ser asignado a cualquier otro tipo de referencia sin cast. Un cast puede ser usado para asignar una variable de referencia de un tipo a otra variable de referencia de un subtipo, pero en tiempo de ejecucion el cast fallara si el objeto en el heap NO ES de tipo compatible con el cast.   Ejemplo: Perro p = (Perro) x.getObject(aPerro); | C:\Users\ingas\Desktop\Atencion2.png |

|  |  |
| --- | --- |
| * Todos los objetos que vienen de un ArrayList<Object> son de tipo Object (esto significa, que solo pueden ser referenciados por una variable de referencia Object, a no ser que uses cast) * La multiple herencia no es permitida en Java, por los problemas asociados con el Deadly Diamond of Death (DDD). Esto significa que tu puedes heredar de una sola clase (tu puedes tener solo una superclase inmediata) * Una interface es como una clase abstracta 100% pura. Esta define solo metodos abstractos. * Crea una interface usando la palabra reservada interface en lugar de class * Implementa una interface usando implements   Ejemplo: Perro implements Mascota   * Tu clase puede implementar multiples interfaces * Una clase que implementa una interface debe implementar todos los metodos de la interface desde todos los metodos de la interface son implicitamente public y abstract. * Para invocar la version de un metodo de una superclase desde una subclase que esta sobreescribiendo el metodo, usamos la palabra reservada super. Ejemplo: s uper.runReport(); | C:\Users\ingas\Desktop\Atencion2.png |