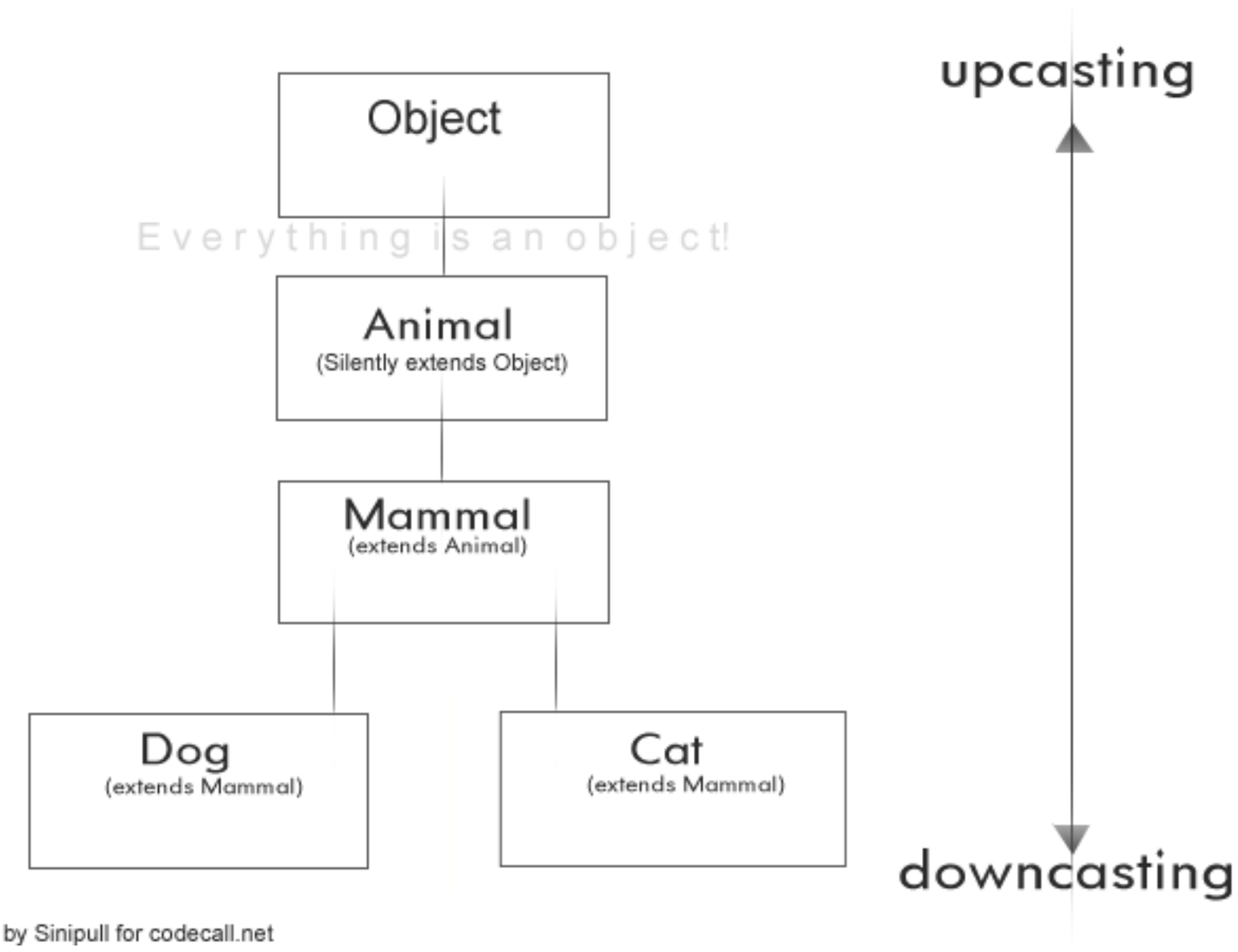
1. Upcasting, downcasting

Upcasting and downcasting are important part of Java, which allow us to build complicated programs using simple syntax, and gives us great advantages, like Polymorphism or grouping different objects. Java permits an object of a subclass type to be treated as an object of any superclass type. This is called upcasting. Upcasting is done automatically, while downcasting must be manually done by the programmer, and i'm going to give my best to explain why is that so.

Upcasting and downcasting are **NOT** like casting primitives from one to other, and i believe that's what causes a lot of confusion, when programmer starts to learn casting objects.

Throughout this tutorial i'm going to use Animal hierarchy to explain how class hierarchy works.

Inheritance



What we have here, is a simplified version of an Animal Hierarchy. You can see, that Cat and Dog are both Mammals, which extends from Animal, which silently extends from Object. By silently, i mean, that Java automatically extends every class from Object class, which isn't extended from something else, so everything is an Object (except primitives).

Now, if you ask - is Cat an Object - It doesn't extend Object, it extends Mammal?
By inheritance Cat gets all the properties its ancestors have. Object is Cat's grandgrandparent, which means Cat is also an Object. Cat is also an Animal and a Mammal, which logically means - if Mammals possess mammary glands and Animals are living beings, then Cat also has mammary glands and is living being.

What this means for a programmer, is that we don't need to write for every possible Animal, that it has health. We just need to write it once, and every Animal gets it through inheritance.
Consider the following example:

Code:

```
class Animal {
        int health = 100;
}

class Mammal extends Animal { }

class Cat extends Mammal { }

class Dog extends Mammal { }

public class Test {
         public static void main(String[] args) {
                 Cat c = new Cat();
                 System.out.println(c.health);
                 Dog d = new Dog();
                 System.out.println(d.health);
```

```
        }
}
```

When running the Test class, it will print "100" and "100" to the console, because both, Cat and Dog inherited the "health" from Animal class.

Upcasting and downcasting

First, you must understand, that by casting you are not actually changing the object itself, you are just labeling it differently.
For example, if you create a Cat and upcast it to Animal, then the object doesn't stop from being a Cat. It's still a Cat, but it's just treated as any other Animal and it's Cat properties are hidden until it's downcasted to a Cat again.
Let's look at object's code before and after upcasting:

Code:

```
Cat c = new Cat();
System.out.println(c);
Mammal m = c; // upcasting
System.out.println(m);

/*
This printed:
    Cat@a90653
    Cat@a90653
*/
```

As you can see, Cat is still exactly the same Cat after upcasting, it didn't change to a Mammal, it's just being labeled Mammal right now. This is allowed, because Cat is a Mammal.

Note that, even though they are both Mammals, Cat cannot be cast to a Dog. Following picture might make it a bit more clear.

Although there's no need to for programmer to upcast manually, it's allowed to do.
Consider the following example:

Code:

```
Mammal m = (Mammal)new Cat();
```

is equal to

Code:

```
Mammal m = new Cat();
```

But downcasting must always be done manually:

Code:

```
Cat c1 = new Cat();
    Animal a = c1;            //automatic upcasting to Animal
    Cat c2 = (Cat) a;    //manual downcasting back to a Cat
```

Why is that so, that upcasting is automatical, but downcasting must be manual? Well, you see, upcasting can never fail. But if you have a group of different Animals and want to downcast them all to a Cat, then there's a chance, that some of these Animals are actually Dogs, and process fails, by throwing ClassCastException.
This is where is should introduce an useful feature called "instanceof", which tests if an object is instance of some Class.
Consider the following example:

Code:

```
Cat c1 = new Cat();
Animal a = c1;                //upcasting to Animal
if(a instanceof Cat){ // testing if the Animal is a Cat
    System.out.println("It's a Cat! Now i can safely downcast it to a Cat, without a fear of failure.");
    Cat c2 = (Cat)a;
}
```

Note, that casting can't always be done in both ways. If you are creating a Mammal, by calling "new Mammal()", you a creating a Object that is a Mammal, but it cannot be downcasted to Dog or Cat, because it's neither of them.
For example:
Code:

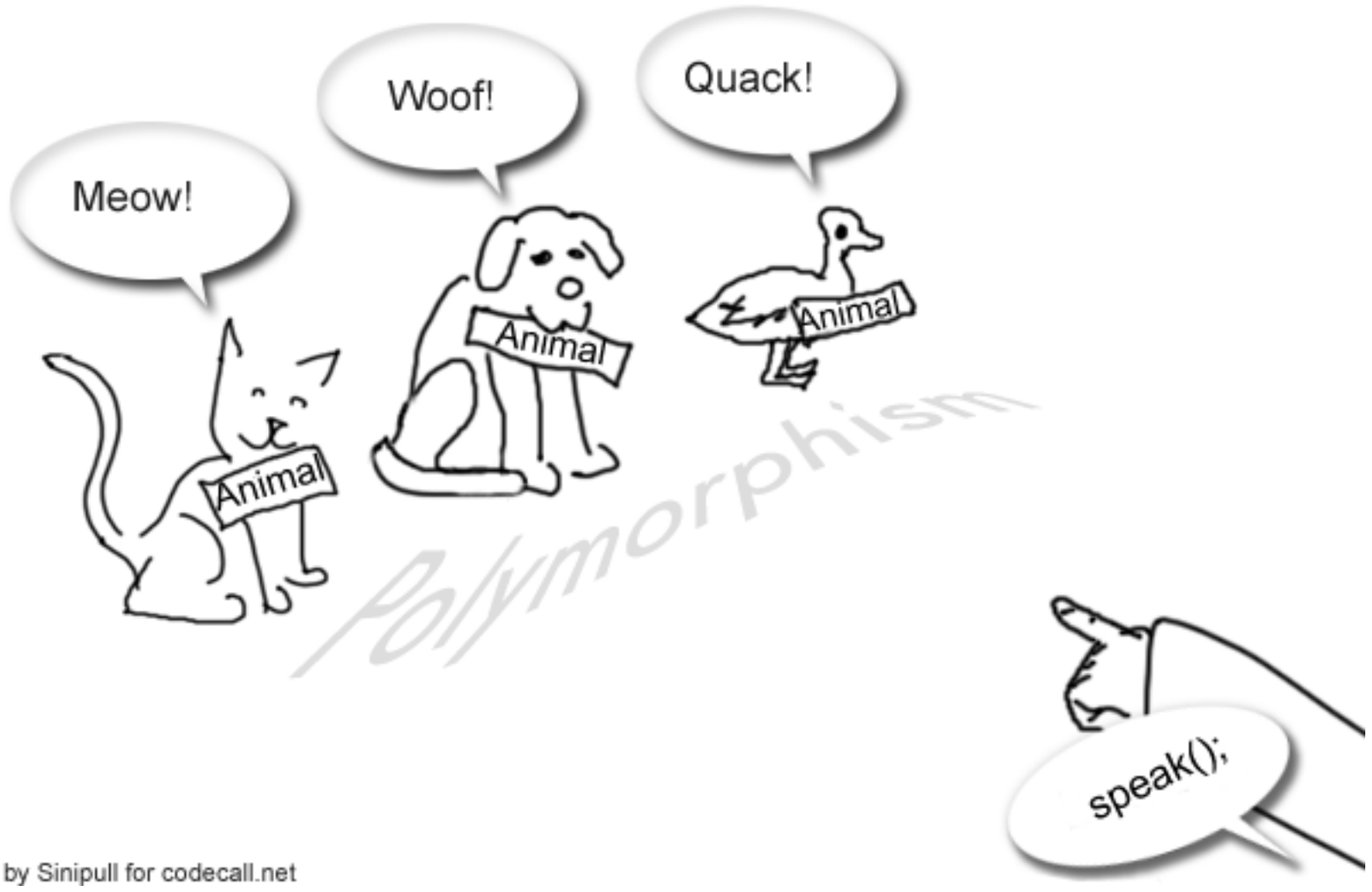
```
Mammal m = new Mammal();
Cat c = (Cat)m;
```

Such code passes compiling, but throws "java.lang.ClassCastException: Mammal cannot be cast to Cat" exception during running, because im trying to cast a Mammal, which is not a Cat, to a Cat.

General idea behind casting, is that, which object is which. You should ask, is Cat a Mammal? Yes, it is - that means, it can be cast.
Is Mammal a Cat? No it isn't - it cannot be cast.
Is Cat a Dog? No, it cannot be cast.
Important: Do not confuse variables with instances here. Cat from Mammal Variable can be cast to a Cat, but Mammal from Mammal variable cannot be cast to a Cat.

Cats cant purr, while being labeled something else
If you upcast an object, it will lose all it's properties, which were inherited from below it's current position. For example, if you cast a Cat to an Animal, it will lose properties inherited from Mammal and Cat. Note, that data will not be lost, you just can't use it, until you downcast the object to the right level.
Why is it like that? If you have a group of Animals, then you can't be sure which ones can meow() and which ones can bark(). That is why you can't make Animal do things, that are only specific for Dogs or Cats.

However the problem above is not an obstacle, if you choose to use polymorphism. Polymorphism uses automatic downcast during method calls. I'm not going to go into details with this one, so i'm referring to Polymorphism tutorial by Turk4n: Polymorphism



Upcasting during method calling

The beauty of casting is that programmer can make general methods, which can take a lot of different classes as an argument. For example:

Code:

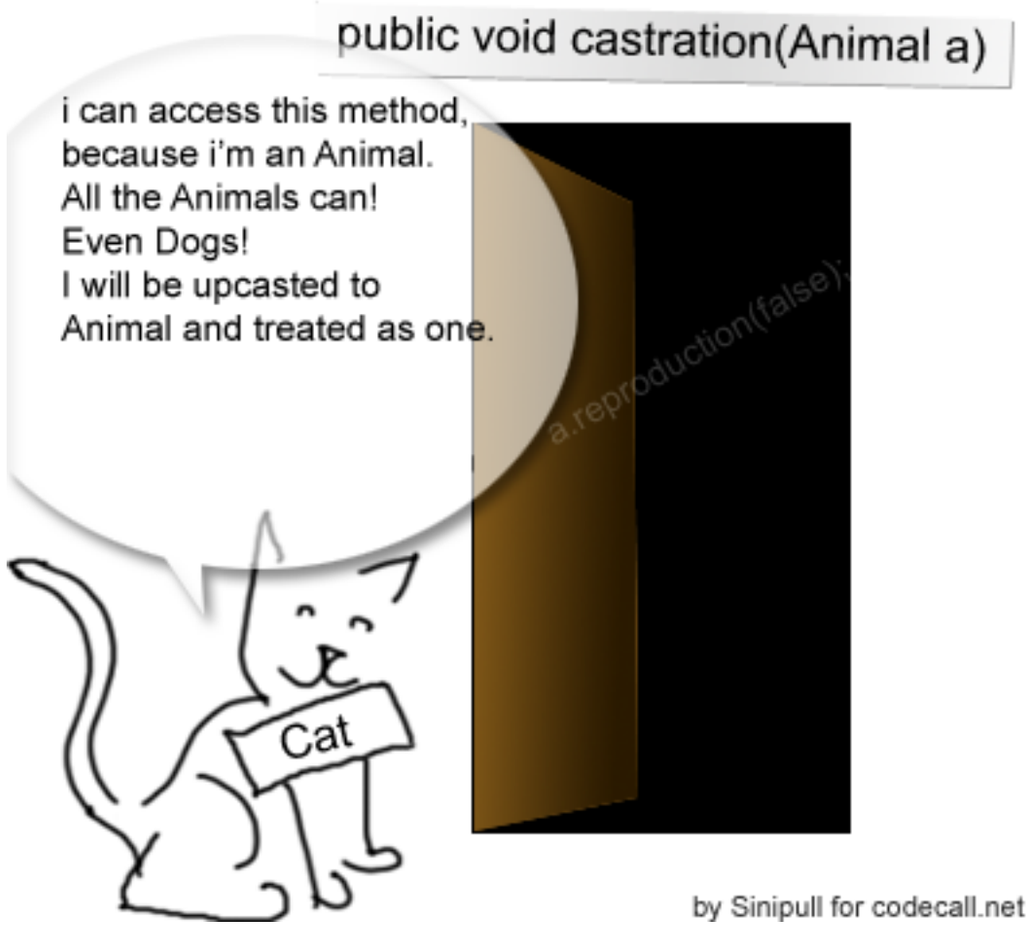
```
public static void stroke(Animal a){
    System.out.println("you stroke the "+a);
}
```

This method can have what ever Animal or it's subclass as an argument. For example calling:

Code:

```
Cat c = new Cat();
Dog d = new Dog();
stroke(c); // automatic upcast to an Animal
stroke(d); // automatic upcast to an Animal
```

..is a correct code.



however, if you have a Cat, that is currently being held by Animal variable, then this variable cannot be argument for a method, that expects only Cats, even though we currently have a instance of Cat - manual downcasting must be done before that.

About variables

Variables can hold instance of objects that are equal or are hierarchically below them. For example Cat c; can hold instances of Cat and anything that is extended from a Cat. Animal can hold Animal, Mammal, etc..
Remember, that instances will always be upcasted to the variable level.

"I really need to make a Dog out of my Cat!"

Well, you can't do it by casting. However, objects are nothing else, but few methods and fields. That means, you can make a new dog out of your Cat's data.

Let's say you have a Cat class:

Code:

```
class Cat extends Mammal {
    Color furColor;
    int numberOfLives;
    int speed;
    int balance;
    int kittens = 0;
```

```
    Cat(Color f, int n, int s, int b){
        this.furColor = f;
        this.numberOfLives = n;
        this.speed = s;
        this.balance = b;
    }
}
```
and a Dog class.

Code:

```
class Dog extends Mammal {
    Color furColor;
    int speed;
    int barkVolume;
    int puppies = 0;

    Dog(Color f, int n, int s, int b){
        this.furColor = f;
        this.speed = s;
        this.barkVolume = b;
    }
}
```
and you want to make a Dog out of the Cat. All you need to do, is, place a method inside of the Cat class, that converts the fields and returns a new Dog based on that.

Code:

```
public Dog toDog(int barkVolume){
    Dog d = new Dog(furColor, speed, barkVolume);
    d.puppies = kittens;
    return d;
}
```

As you can see, they don't match that well, so some fields were inconvertible, and some data had to be made from scratch. Notice, that numberOfLives and Balance were not converted, and barkVolume was completely new data. If you have 2 Classes, that match perfectly, then hurray, but it rarely happens.
conversion can now be called from where ever you need:

Code:

```
Cat c = new Cat(Color.black, 9, 20, 40);
Dog d = c.toDog(50);
```

Thanks for reading.