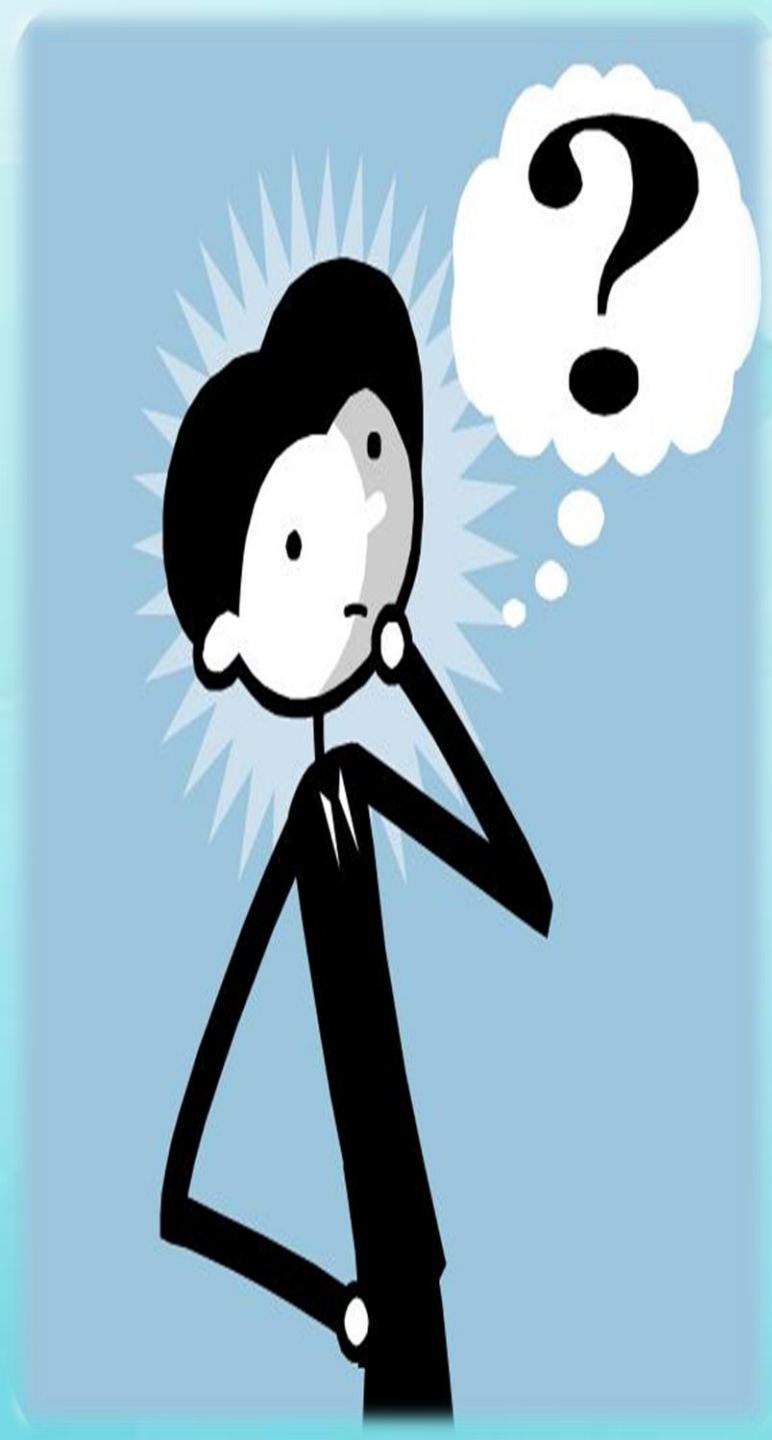
Pemrograman Berorientasi Objek
Polimorphisme dan Serious Polimorphisme

Adam Mukharil Bachtiar
Teknik Informatika UNIKOM



Polimorphisme dan Serious Polimorphisme

1. Definisi Polimorphisme
2. Overloading Method
3. Overriding Method
4. Kata Kunci Final
5. Abstract Class
6. Interface

Definisi Polimorphisme

Suatu aksi yang memungkinkan pemrogram menyampaikan pesan tertentu keluar dari hirarki obyeknya, dimana obyek yang berbeda memberikan tanggapan/respon terhadap pesan yang sama sesuai dengan sifat masing-masing obyek.

Gambaran Polimorphisme

Proses Polimorphisme

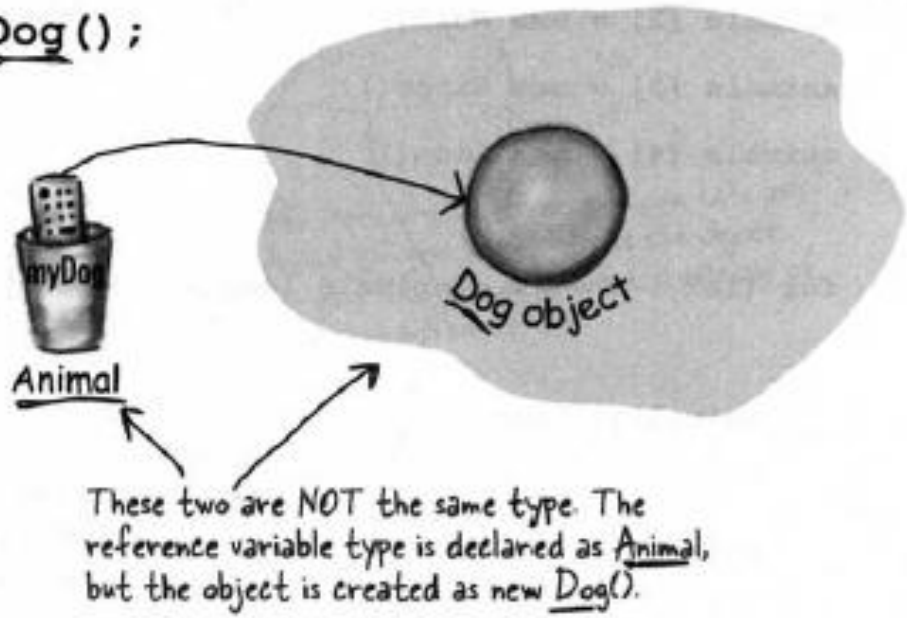
1. Overloading Method

2. Overriding method

Objek Polimorphisme (1)

with polymorphism, the reference and the object can be *different*.

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



Objek Polimorphisme (2)

OK, OK maybe an example will help.

```
Animal[] animals = new Animal[5];
animals [0] = new Dog();
animals [1] = new Cat();
animals [2] = new Wolf();
animals [3] = new Hippo();
animals [4] = new Lion();

for (int i = 0; i < animals.length; i++) {
    animals[i].eat();
    animals[i].roam();
}
```

Declare an array of type Animal. In other words, an array that will hold objects of type Animal.

But look what you get to do... you can put ANY subclass of Animal in the Animal array!

And here's the best polymorphic part (the raison d'être for the whole example), you get to loop through the array and call one of the Animal-class methods, and every object does the right thing!

When 'i' is 0, a Dog is at index 0 in the array, so you get the Dog's eat() method. When 'i' is 1, you get the Cat's eat() method

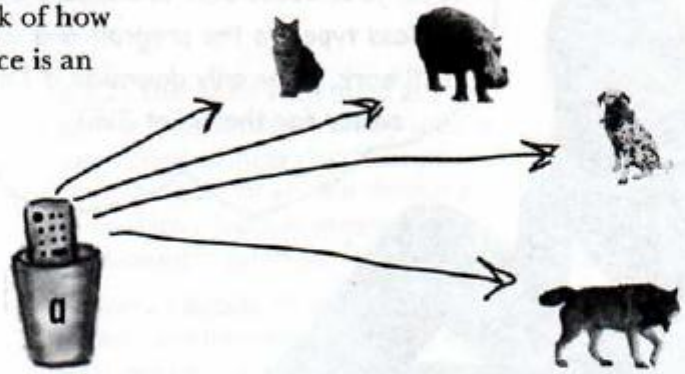
Same with roam().

Objek Polimorphisme (3)

But wait! There's more!

You can have polymorphic arguments and return types.

If you can declare a reference variable of a supertype, say, Animal, and assign a subclass object to it, say, Dog, think of how that might work when the reference is an argument to a method...




```
class Vet {

   public void giveShot(Animal a) {

     // do horrible things to the Animal at

     // the other end of the 'a' parameter

     a.makeNoise();

   }

}
```

The Animal parameter can take ANY Animal type as the argument. And when the Vet is done giving the shot, it tells the Animal to makeNoise(), and whatever Animal is really out there on the heap, that's whose makeNoise() method will run.

Objek Polimorphisme (4)

```
class PetOwner {
   public void start() {
      Vet v = new Vet();
      Dog d = new Dog();
      Hippo h = new Hippo();
      v.giveShot(d);
      v.giveShot(h);
   }
}
```

The Vet's giveShot() method can take any Animal you give it. As long as the object you pass in as the argument is a subclass of Animal, it will work.

Dog's makeNoise() runs

Hippo's makeNoise() runs

Definisi Overloading Method (1)

Overloading adalah keadaan di mana beberapa method yang terdapat pada suatu class memiliki nama yang sama dengan fungsionalitas yang sama.

Definisi Overloading Method (2)

An overloaded method is just a different method that happens to have the same method name. It has nothing to do with inheritance and polymorphism. An overloaded method is NOT the same as an overridden method.

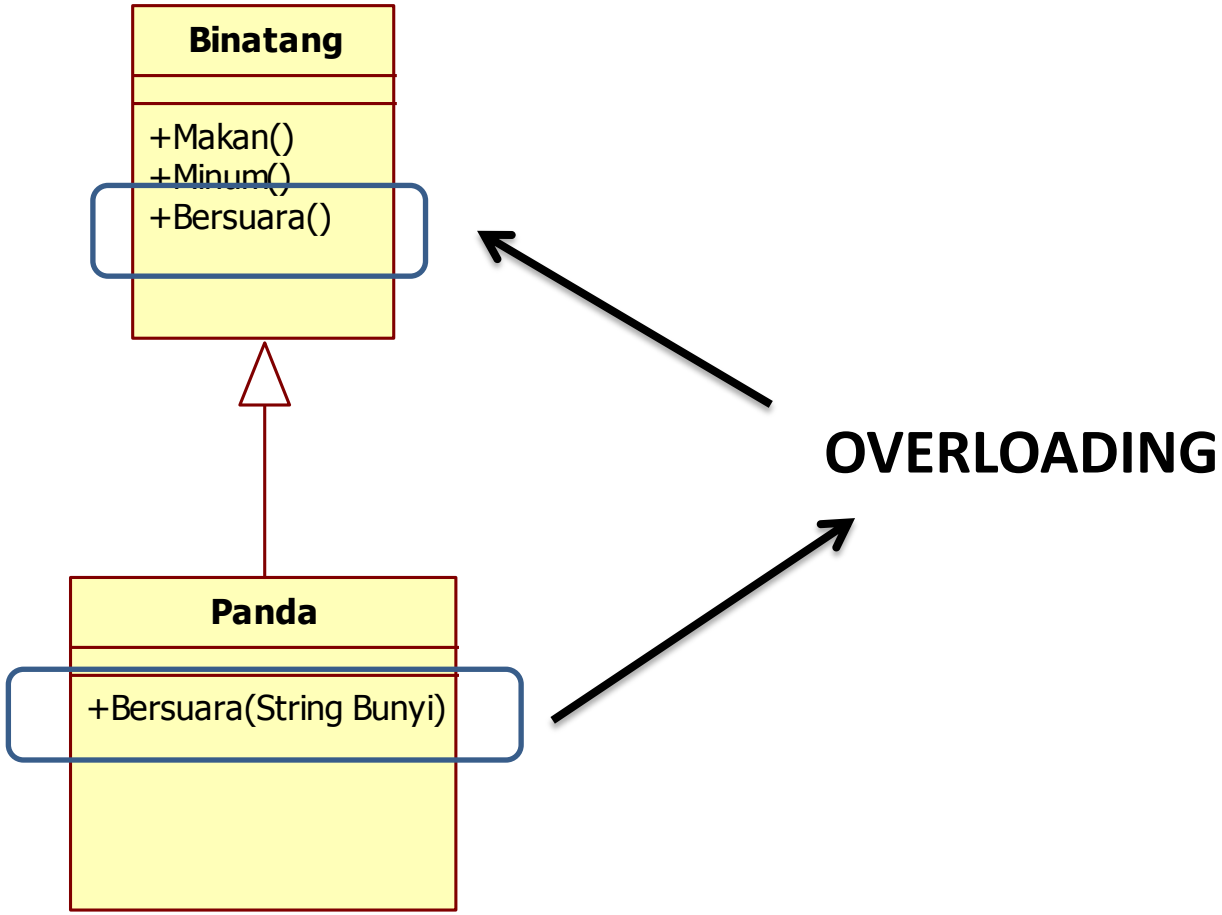
Definisi Overloading Method (3)

Sebagai contoh:

titik(x,y) → titik dipandang dua dimensi

titik(x,y,z) → titik dipandang tiga dimensi

Class Diagram Overloading



Aturan Overloading Method

- **The return types can be different.**

  You're free to change the return types in overloaded methods, as long as the argument lists are different.

- **You can't change ONLY the return type.**

  If only the return type is different, it's not a valid over*load*—the compiler will assume you're trying to over*ride* the method. And even *that* won't be legal unless the return type is a subtype of the return type declared in the superclass. To overload a method, you MUST change the argument list, although you *can* change the return type to anything.

- **You *can* vary the access levels in any direction.**

  You're free to overload a method with a method that's more restrictive. It doesn't matter, since the new method isn't obligated to fulfill the contract of the overloaded method.

Penggalan Overloading Method

```
public class Overloads {

   String uniqueID;

   public int addNums(int a, int b) {
      return a + b;
   }

   public double addNums(double a, double b) {
       return a + b;
   }

   public void setUniqueID(String theID) {
      // lots of validation code, and then:
      uniqueID = theID;
   }

   public void setUniqueID(int ssNumber) {
      String numString = "" + ssNumber;
      setUniqueID(numString);
   }
}
```

Contoh Overloading Method (JAVA)

```
public class OrangTua {
    private int umur;

    public int getUmur() {
        return umur;
    }

    public void setUmur(int umur) {
        this.umur = umur;
    }

    public void tampilUmur(){
        System.out.println("Umur Orang Tua : "+umur);
    }
}
```

Contoh Overloading Method (JAVA)

```
public class Anak extends OrangTua {
    private int umur2;

    public int getUmur2() {
        return umur2;
    }

    public void setUmur2(int umur2) {
        this.umur2 = umur2;
    }

    //overloading
    public void tampilUmur(int a){
        System.out.println("Umur Anak : "+umur2);
    }
}
```

Contoh Overloading Method (JAVA)

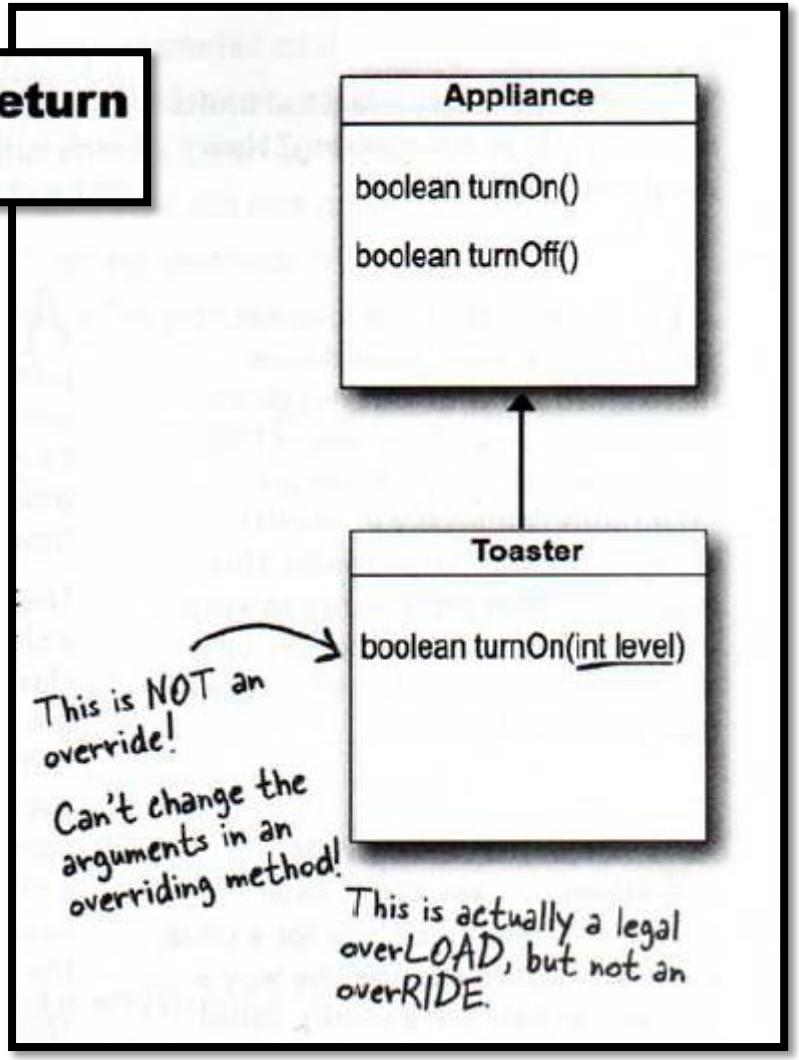
```
public class TesterPolimorphisme {

    public static void main(String[] args) {
        Anak x = new Anak();
         x.setUmur(40);
         x.setUmur2(11);
         x.tampilUmur(1);//Pemanggilan Overloading
    }
}
```

Definisi Overriding Method

Overriding adalah keadaan di mana suatu method di sub class mengingkari method yang ada pada pada super classnya.

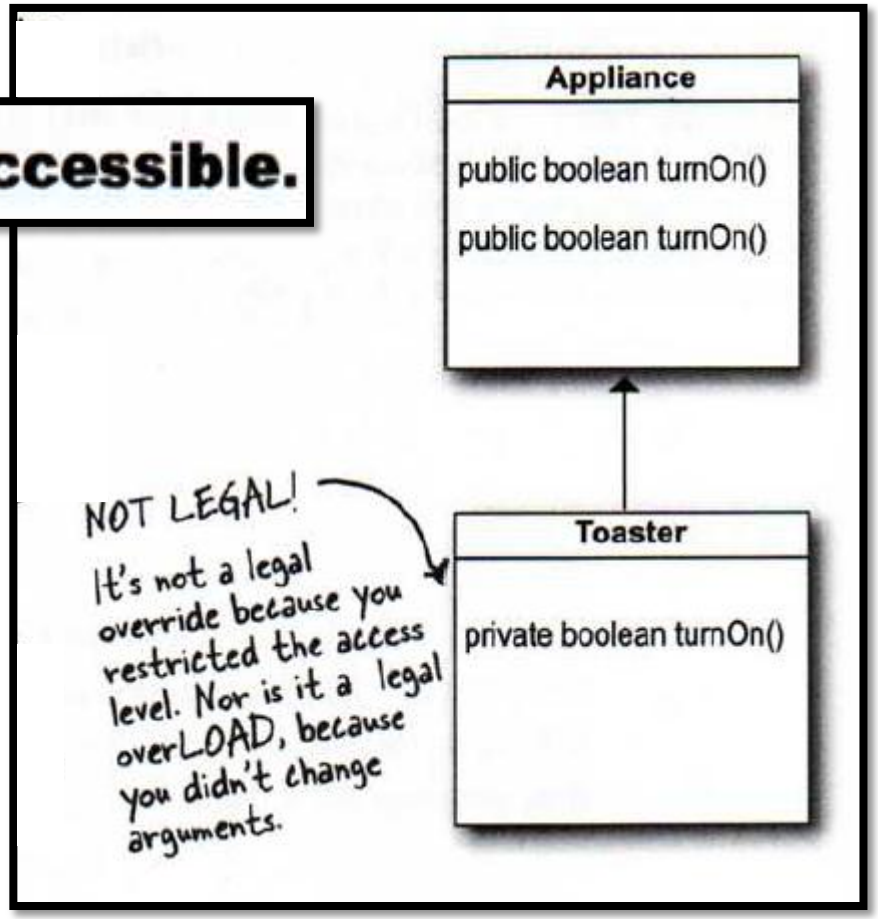
Aturan Overriding Method (1)

1. **Arguments must be the same, and return types must be compatible.**

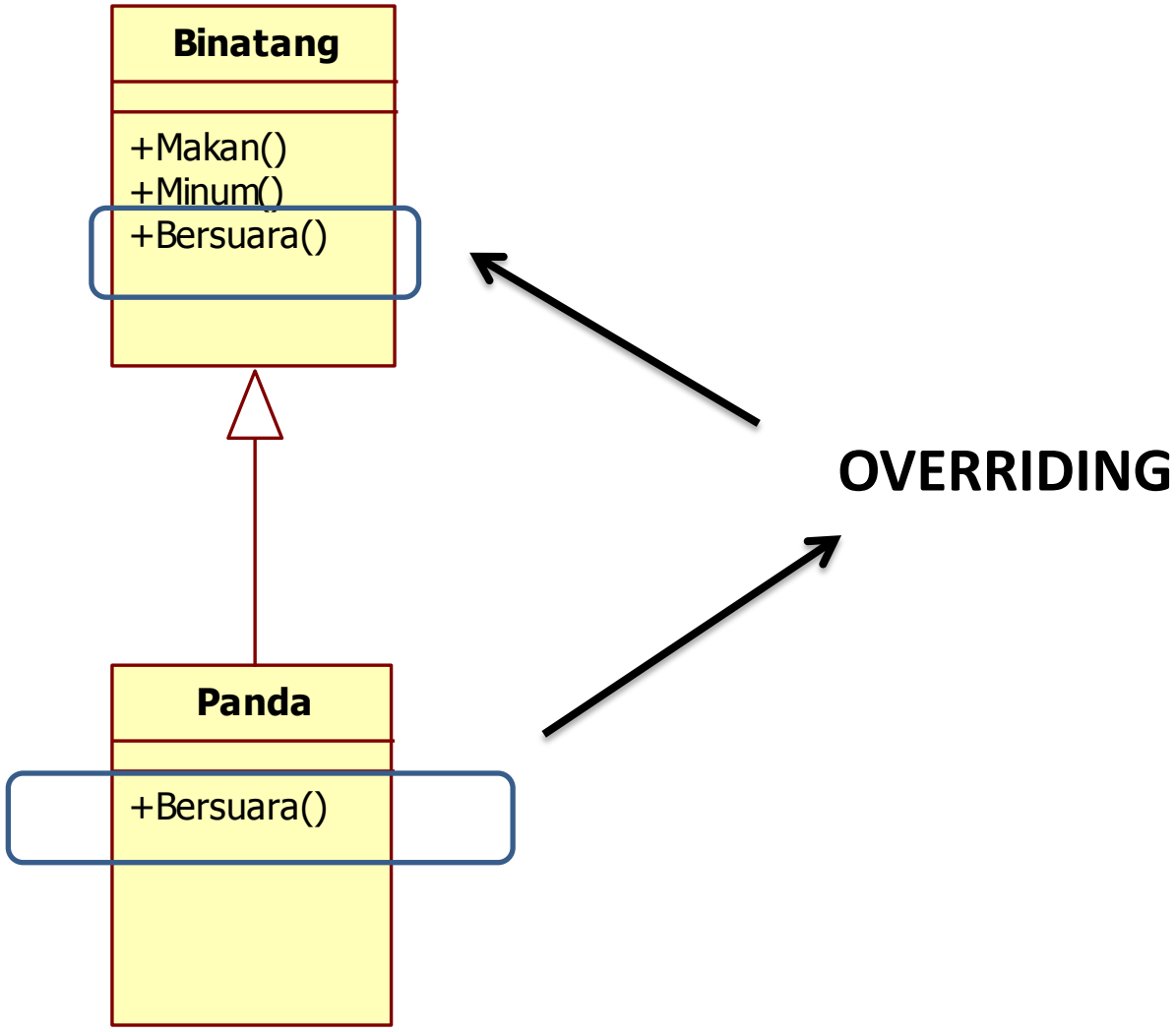


Aturan Overriding Method (2)

② The method can't be less accessible.



Class Diagram Overloading



Contoh Overriding Method (JAVA)

```
public class OrangTua {
    private int umur;

    public int getUmur() {
        return umur;
    }

    public void setUmur(int umur) {
        this.umur = umur;
    }

    public void tampilUmur(){
        System.out.println("Umur Orang Tua : "+umur);
    }
}
```

Contoh Overriding Method (JAVA)

```
public class Anak extends OrangTua {
    private int umur2;

    public int getUmur2() {
        return umur2;
    }

    public void setUmur2(int umur2) {
        this.umur2 = umur2;
    }

    @Override //Penanda Overriding di java
    public void tampilUmur(){
        super.tampilUmur();
        System.out.println("Umur Anak : "+umur2);
    }
}
```

Contoh Overriding Method (JAVA)

```
public class Anak extends OrangTua {
    private int umur2;

    public int getUmur2() {
        return umur2;
    }

    public void setUmur2(int umur2) {
        this.umur2 = umur2;
    }

    @Override //Penanda Overriding di java
    public void tampilUmur(){
        super.tampilUmur();
        System.out.println("Umur Anak : "+umur2);
    }
}
```

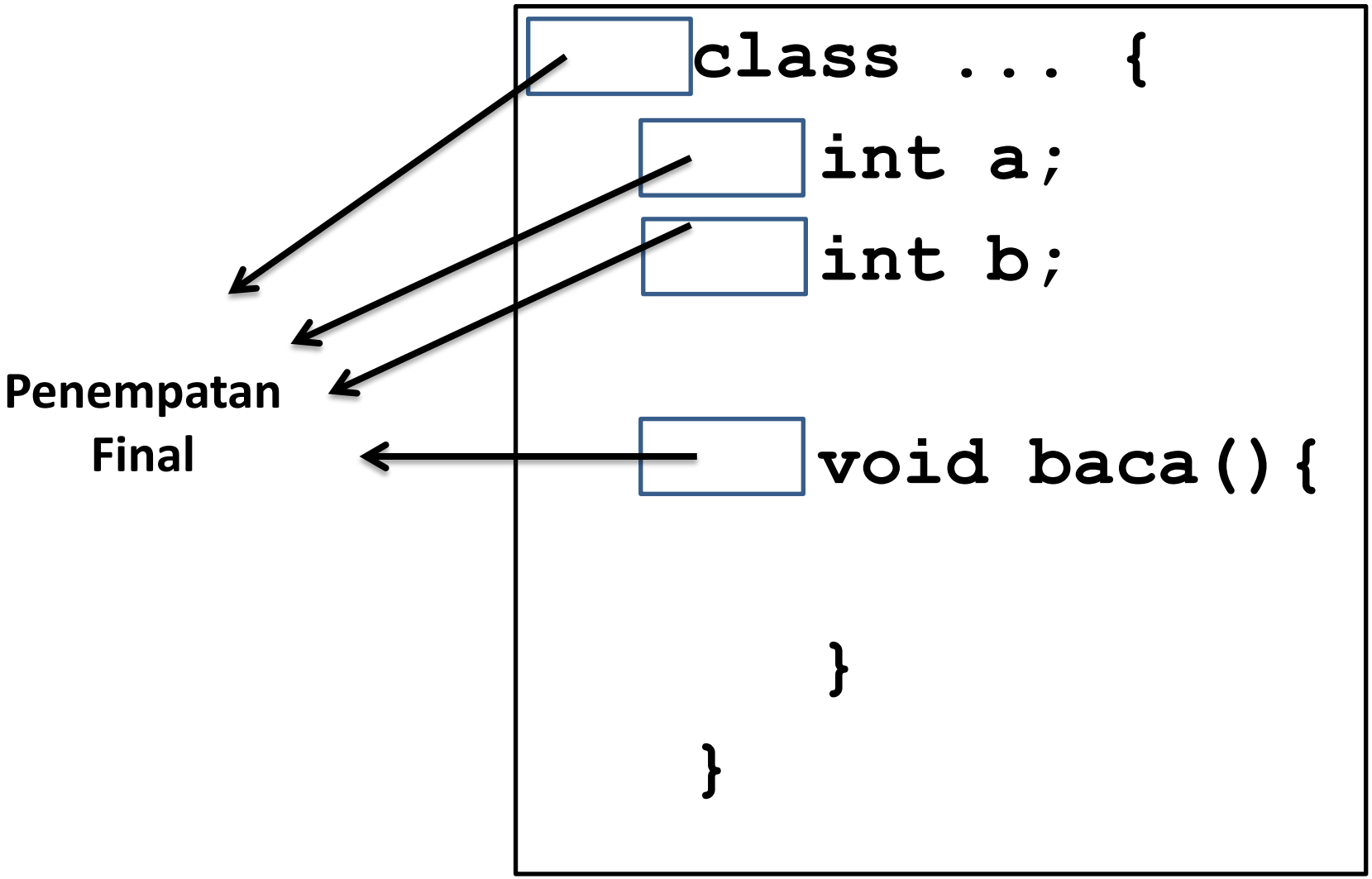
Contoh Overriding Method (JAVA)

```
public class TesterPolimorphisme {

    public static void main(String[] args) {
        Anak x = new Anak();
         x.setUmur(40);
         x.setUmur2(11);
         x.tampilUmur();//Pemanggilan Overriding
    }
}
```

Kata Kunci Final

Jika suatu class ataupun atribut serta method di dalamnya tidak ingin di-extend lagi oleh suatu subclass maka gunakan kata kunci final. Kata kunci final biasanya digunakan untuk menunjukkan sub class terakhir dari suatu inheritance.

Penempatan Kata Kunci Final



FAQ Kata Kunci Final

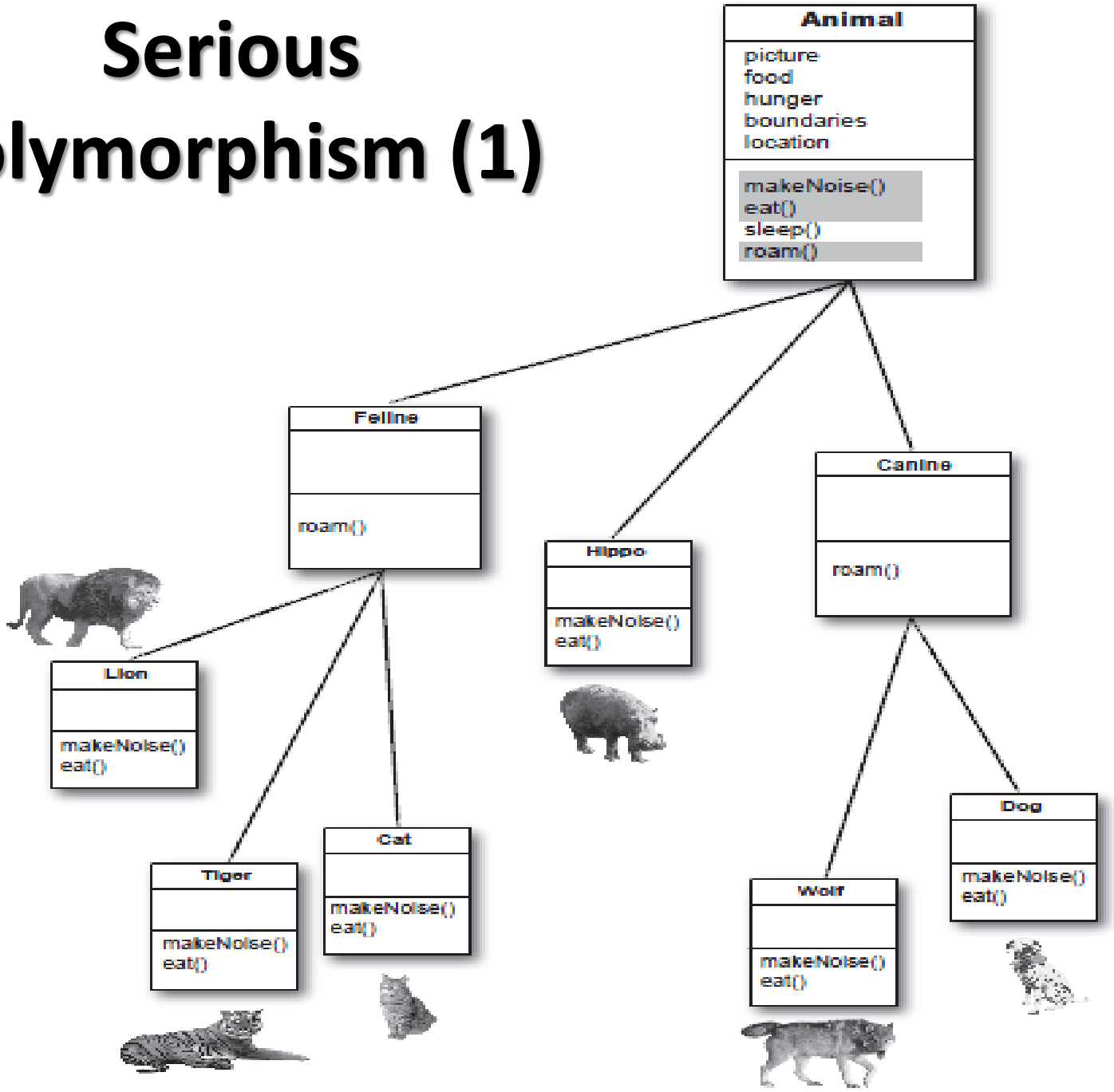
Q: Why would you ever want to make a final class? What advantage would there be in preventing a class from being subclassed?

A: Typically, you won't make your classes final. But if you need security — the security of knowing that the methods will always work the way that you wrote them (because they can't be overridden), a final class will give you that. A lot of classes in the Java API are final for that reason. The String class, for example, is final because, well, imagine the havoc if somebody came along and changed the way Strings behave!

Q: Can you make a *method* final, without making the whole *class* final?

A: If you want to protect a specific method from being overridden, mark the *method* with the `final` modifier. Mark the whole *class* as final if you want to guarantee that *none* of the methods in that class will ever be overridden.

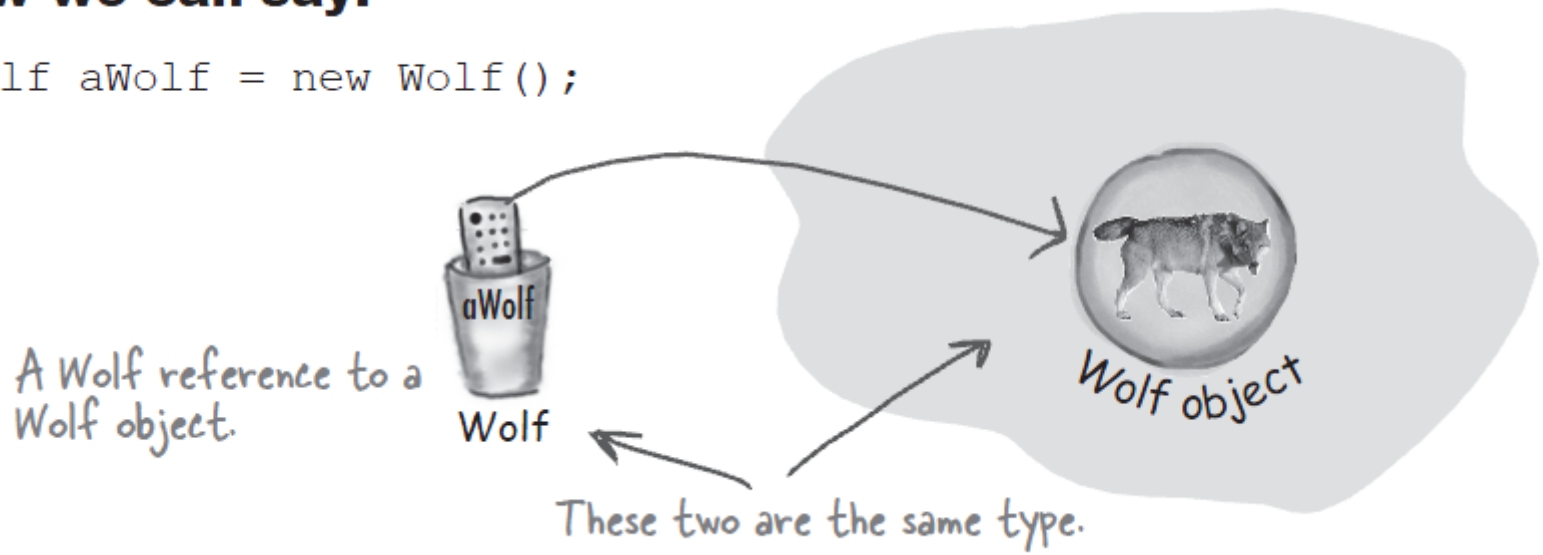
Serious Polymorphism (1)



Serious Polymorphism (2)

We know we can say:

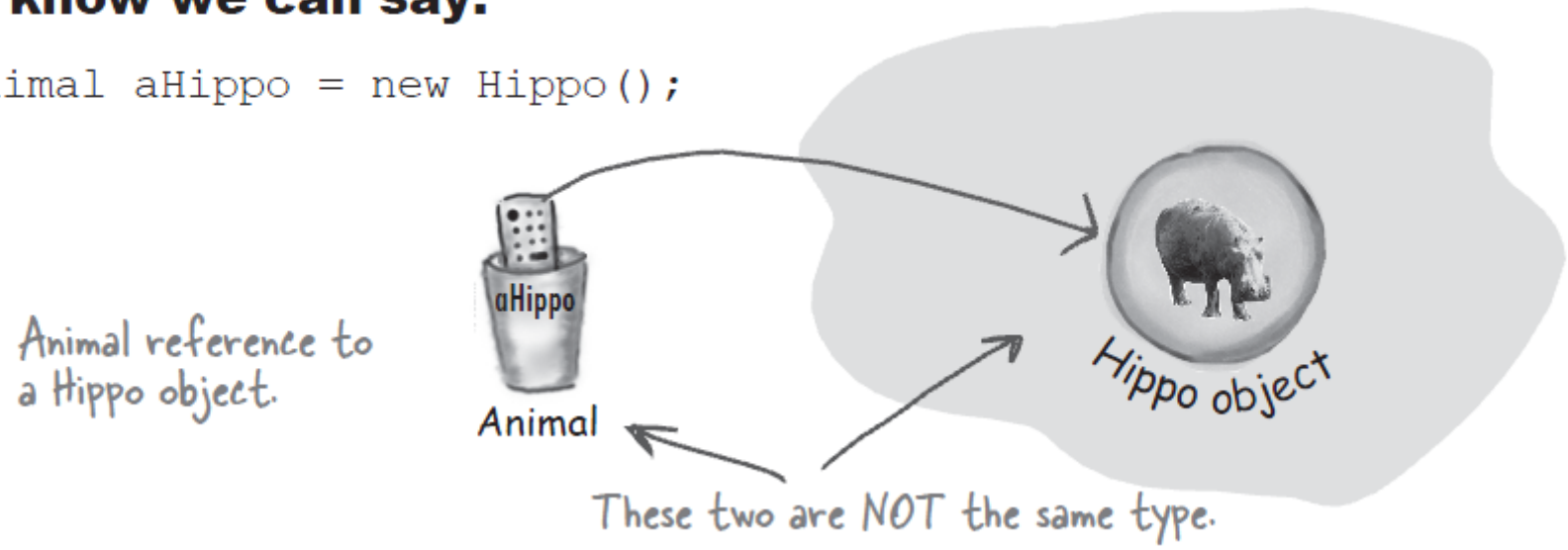
```
Wolf aWolf = new Wolf();
```



Serious Polymorphism (3)

And we know we can say:

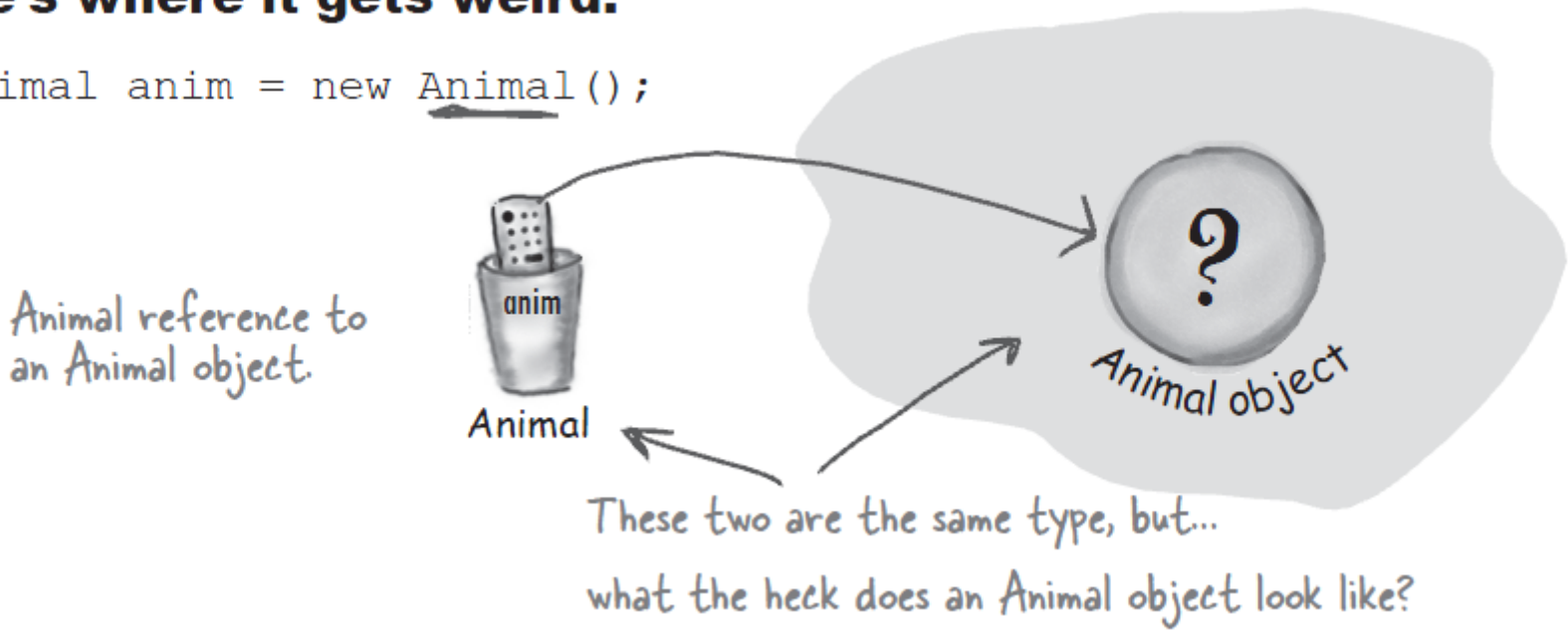
```
Animal aHippo = new Hippo();
```



Serious Polymorphism (4)

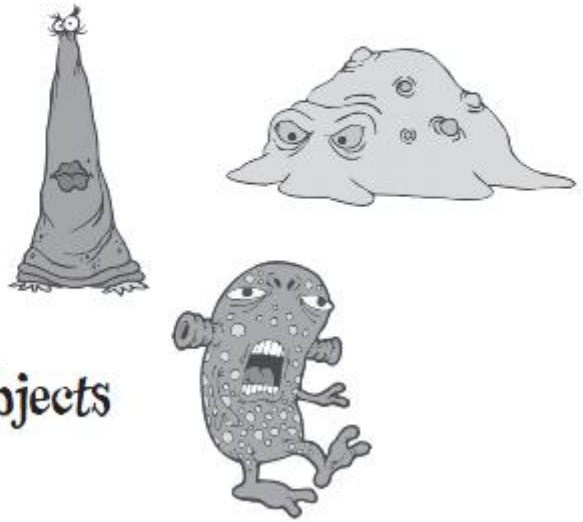
But here's where it gets weird:

```
Animal anim = new Animal();
```



Serious Polymorphism (5)

What does a new Animal() object *look* like?



What are the instance variable values?

Some classes just should *not* be instantiated!

Abstract Class (1)

Format class yang tidak bisa diinstansiasi secara langsung. Kelas ini harus diimplemantasi terlebih dahulu sebelum dapat digunakan.

Abstract Class (2)

Format pembuatan:

```
abstract class <nama_class> {

}
```

Contoh:

```
abstract class animal{

}
```

Abstract Class (3)

Format penggunaan:

```
class <nama_class> extends <nama_class_abstract>{

}
```

Contoh:

```
class Wolf extends Animal {

}
```

Abstract Class (4)

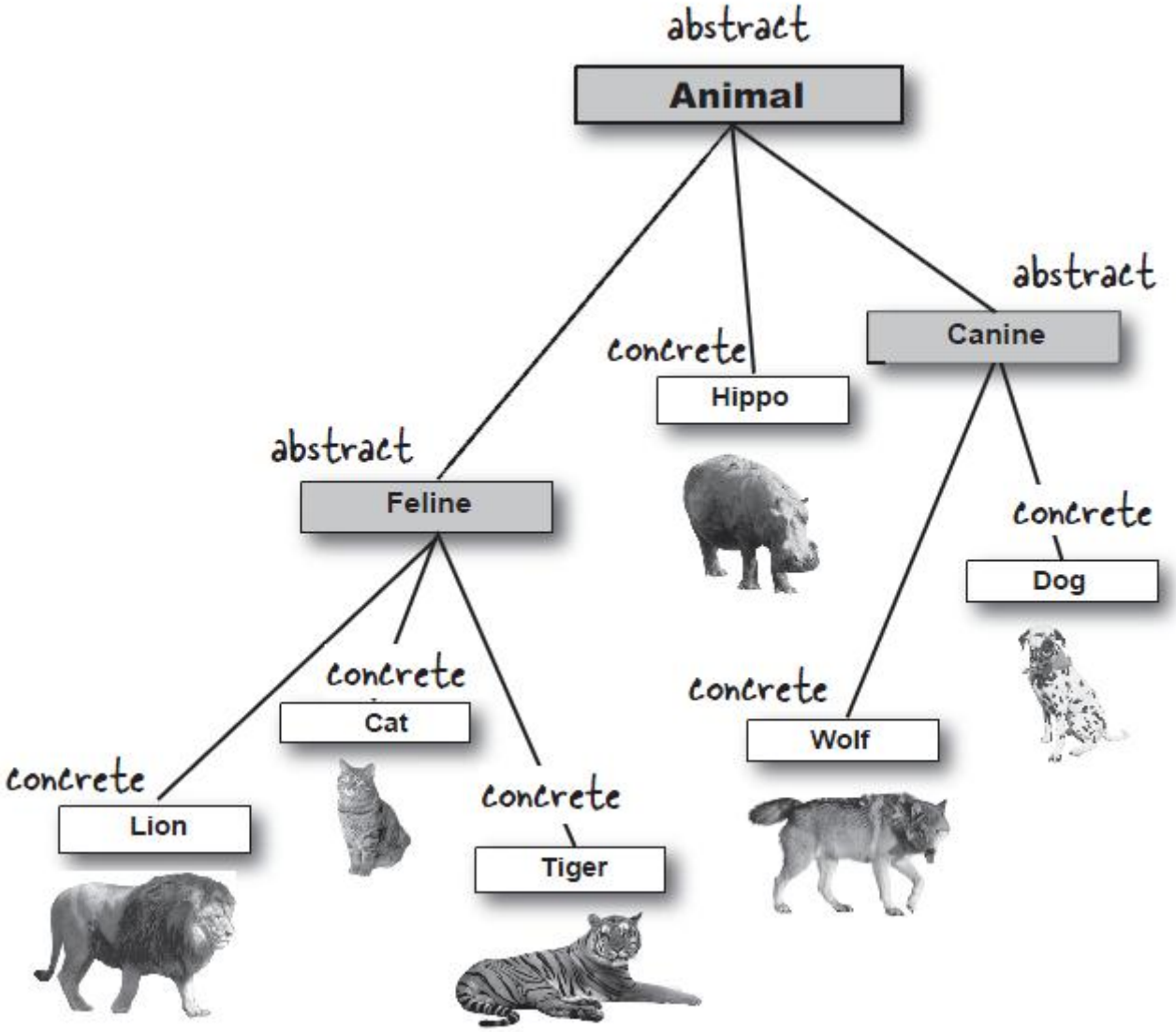
```
abstract public class Canine extends Animal
{
    public void roam() { }
}
```

```
public class MakeCanine {
    public void go() {
        Canine c;
        c = new Dog();
        c = new Canine();
        c.roam();
    }
}
```

This is OK, because you can always assign a subclass object to a superclass reference, even if the superclass is abstract.

class Canine is marked abstract, so the compiler will NOT let you do this.

Abstract Class VS Concrete Class



Abstract Method (1)

Method yang hanya berupa prototipe dan untuk bagian isi method akan diimplementasi pada class yang lain.

Abstract Class (2)

Format:

```
<hak_akses> abstract void <nama_method>() ;
```

Contoh:

```
Public abstract void roam();
```

Abstract Class (3)

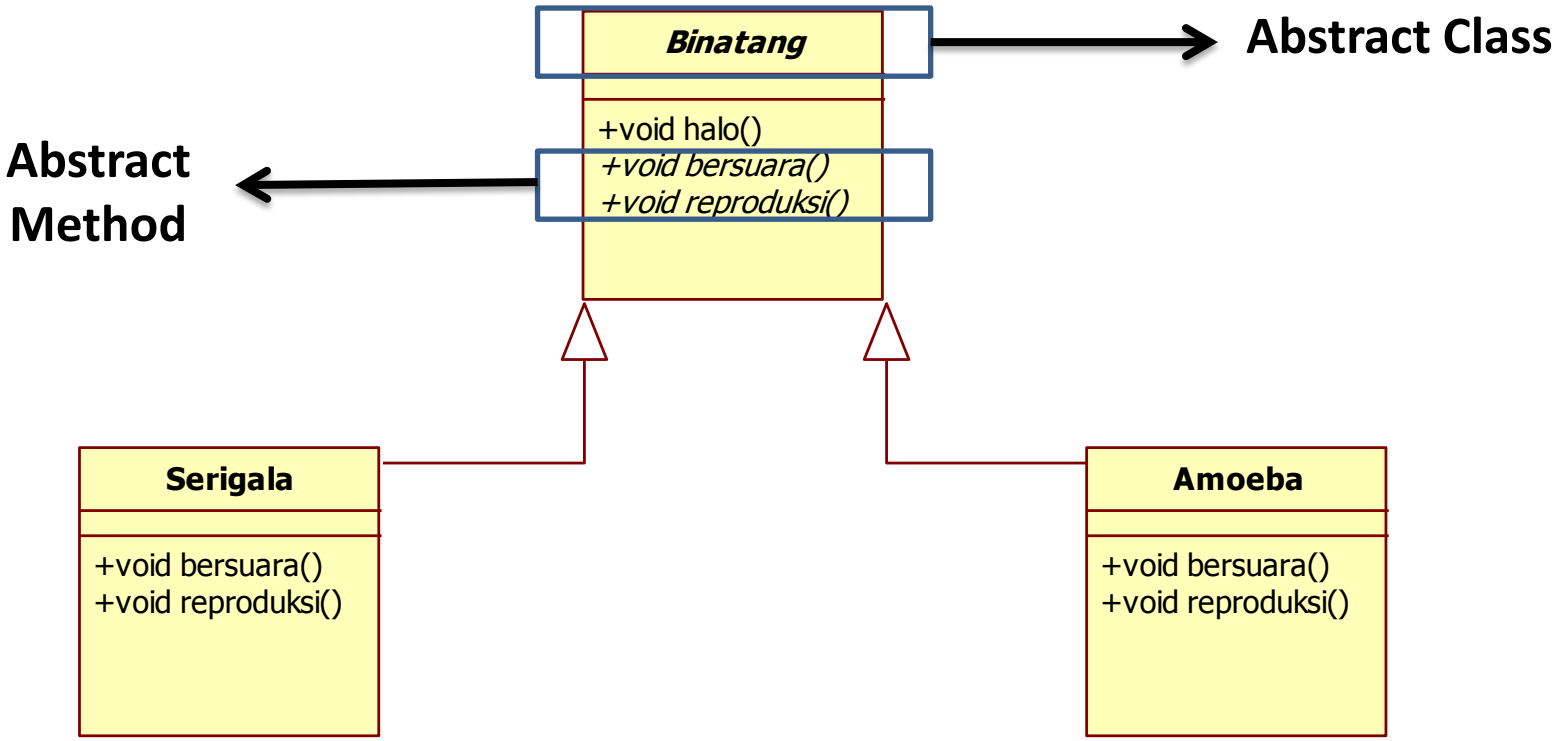
An abstract method has no body!

If you declare an abstract *method*, you MUST mark the *class* abstract as well. You can't have an abstract method in a non-abstract class.

***Implementing* an abstract method is just like *overriding* a method.**

You MUST implement all abstract methods

Perancangan Abstract Class



Contoh Abstract Class (1)

```
public abstract class Binatang {

    public void halo(){

        System.out.println("Halo saya binatang!");

    }



    public abstract void bersuara();

    public abstract void reproduksi();

}
```

Contoh Abstract Class (2)

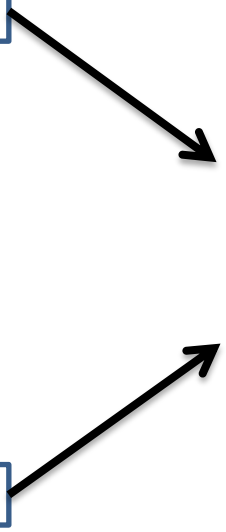
```
public class Serigala extends Binatang{

    @Override

    public void bersuara() {

        System.out.println("Auuuuwwww!!!!");

    }


    @Override

    public void reproduksi() {

        System.out.println("Reproduksi generatif!");

    }

}
```

Contoh Abstract Class (3)

```
public class Amoeba extends Binatang{

    @Override
    public void bersuara() {
        System.out.println("Tidak bersuara");
    }


    @Override
    public void reproduksi() {
        System.out.println("Reproduksi vegetatif!");
    }
}
```

Contoh Abstract Class (4)

```
public class TesterAbstract {
     public static void main(String[] args) {
        Binatang x=new Serigala();
        x.halo();
        x.bersuara();
        x.reproduksi();

        System.out.println("");

        Amoeba y=new Amoeba();
        y.halo();
        y.bersuara();
        y.reproduksi();
    }
}
```



Apa Bedanya?

Interface (1)

Sebuah blok program yang hanya berisi method-method untuk diimplementasi di kelas yang lain. Class implementasi bisa mengimplementasi lebih dari satu interface. Interface diimplementasikan bukan di-extends

Interface (2)

Format pembuatan:

```
<hak_akses> interface <nama_interface> {

}
```

Contoh:

```
public interface animal{

}
```

Interface (3)

Format penggunaan:

```
class <nama_class> implements <nama_interface>{

}
```

Contoh:

```
Class Wolf implements Animal{

}
```

Interface to the rescue!

It's called "multiple inheritance" and it can be a Really Bad Thing.

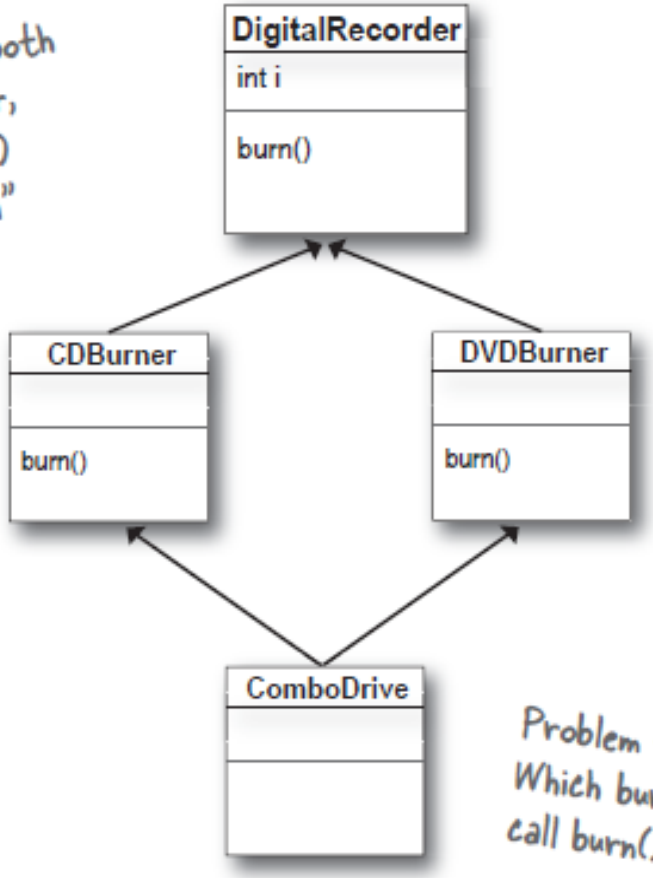
That is, if it were possible to do in Java.

But it isn't, because multiple inheritance has a problem known as The Deadly Diamond of Death.

A Java interface is like a 100% pure abstract class.

All methods in an interface are abstract, so any class that IS-A Pet MUST implement (i.e. override) the methods of Pet.

Deadly Diamond of Death



CDBurner and DVDBurner both inherit from DigitalRecorder, and both override the burn() method. Both inherit the "i" instance variable.

Imagine that the "i" instance variable is used by both CDBurner and DVDBurner, with different values. What happens if ComboDrive needs to use both values of "i"?

Problem with multiple inheritance. Which burn() method runs when you call burn() on the ComboDrive?

Kenapa Interface?

Making and Implementing the Pet interface

interface methods are implicitly public and abstract, so typing in 'public' and 'abstract' is optional (in fact, it's not considered 'good style' to type the words in, but we did here just to reinforce it, and because we've never been slaves to fashion...)

You say 'interface' instead of 'class' here

```
public interface Pet {

    public abstract void beFriendly();

    public abstract void play();

}
```

All interface methods are abstract, so they MUST end in semicolons. Remember, they have no body!

Dog IS-A Animal and Dog IS-A Pet

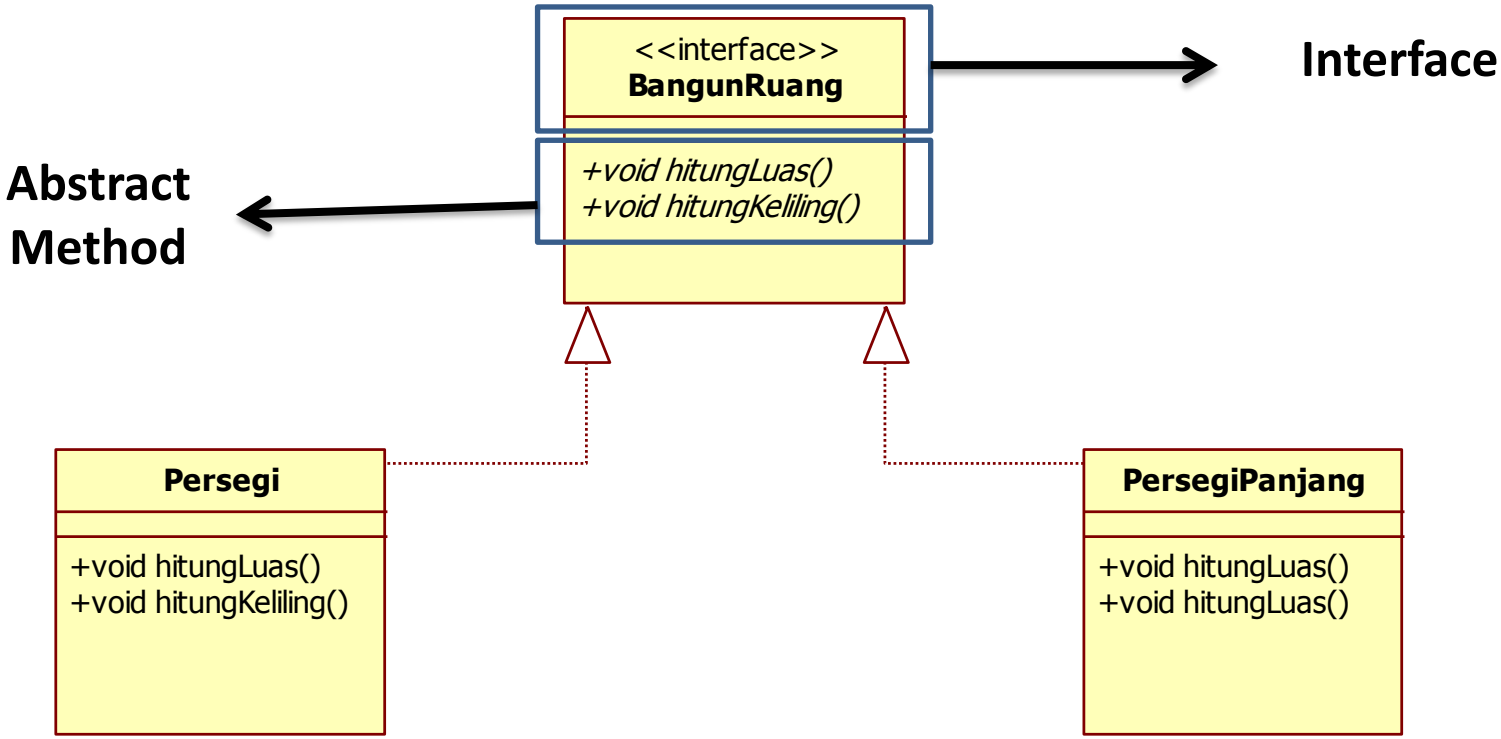
You say 'implements' followed by the name of the interface.

```
public class Dog extends Canine implements Pet {

   public void beFriendly() {...}

   public void play() {..}


   public void roam() {...}

   public void eat() {...}

}
```

You SAID you are a Pet, so you MUST implement the Pet methods. It's your contract. Notice the curly braces instead of semicolons.

These are just normal overriding methods.

Perancangan Interface



Contoh Interface (1)

```
public interface BangunRuang {

    public void hitungLuas();

    public void hitungKeliling();

}
```

Contoh Interface (2)

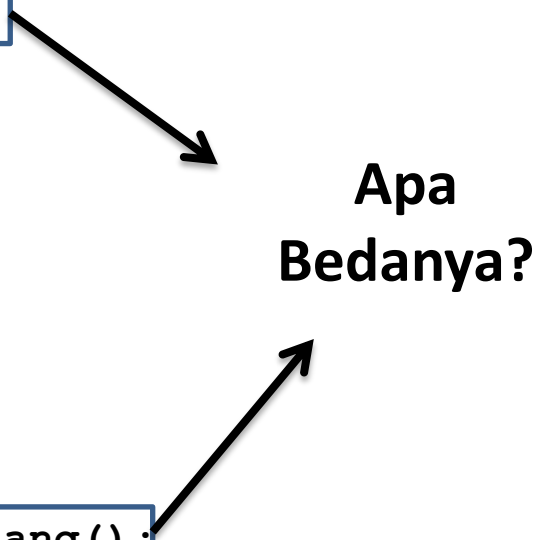
```
public class Persegi implements BangunRuang{
    private int sisi=4;
    private int luas;
    private int kel;

    @Override
    public void hitungLuas() {
        luas=sisi*sisi;
        System.out.println("Luas Persegi      : "+luas);
    }

    @Override
    public void hitungKeliling() {
        kel=4*sisi;
        System.out.println("Keliling Persegi : "+kel);
    }
}
```

Contoh Interface (3)

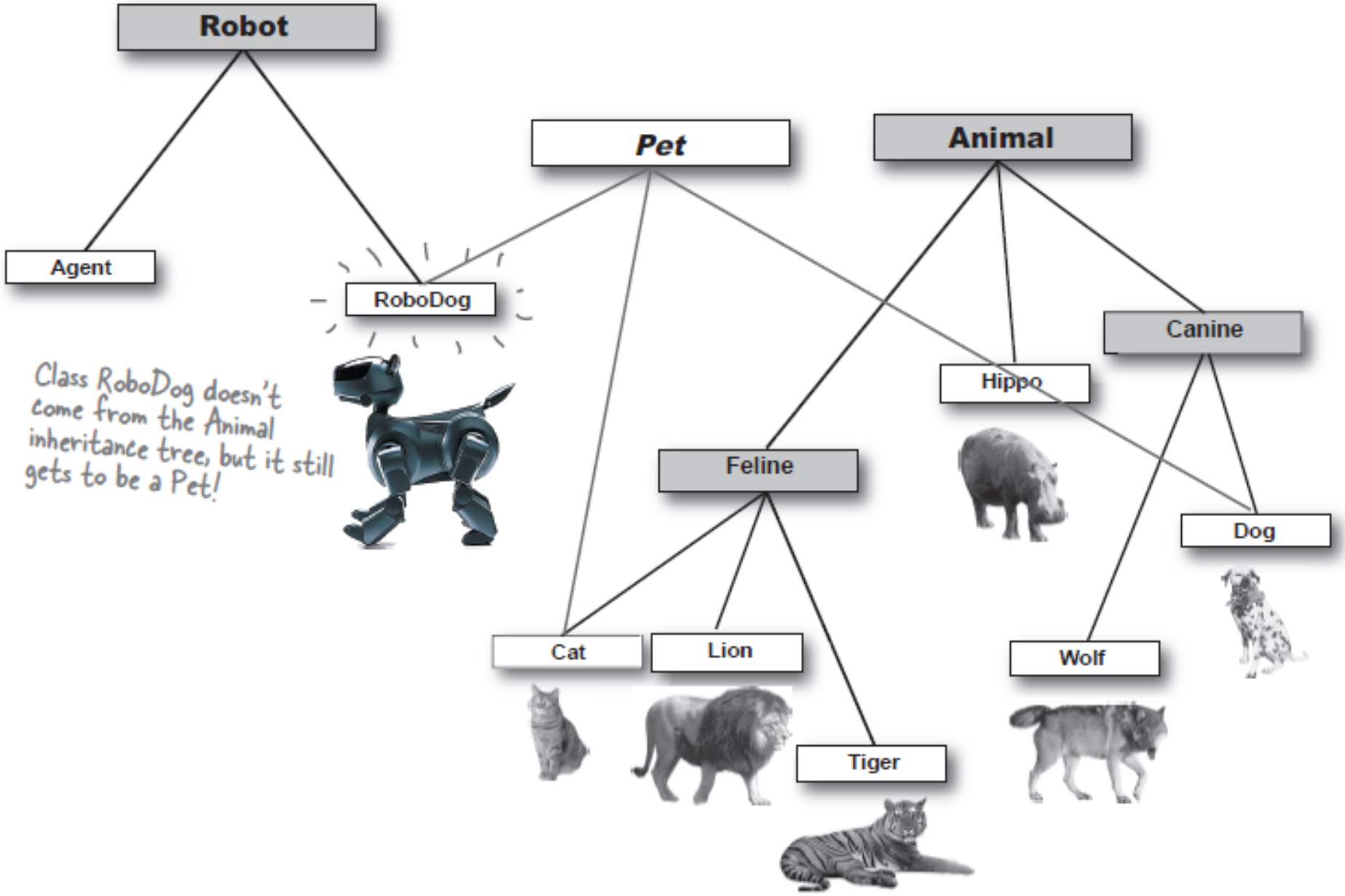
```
public class PersegiPanjang implements BangunRuang{
    int panjang=4;
    int lebar=3;
    private int luas;
    private int kel;

    @Override
    public void hitungLuas() {
        luas=panjang*lebar;
        System.out.println("Luas Persegi Panjang      : "+luas);
    }

    @Override
    public void hitungKeliling() {
        kel=(2*panjang)+(2*lebar);
        System.out.println("Keliling Persegi Panjang  : "+kel);
    }
}
```

Contoh Interface (4)

```
public class TesterInterface {

    public static void main(String[] args) {

        Persegi x=new Persegi();

        x.hitungKeliling();

        x.hitungLuas();



        System.out.println();



        BangunRuang y=new PersegiPanjang();

        y.hitungKeliling();

        y.hitungLuas();

    }

}
```



Classes from *different* inheritance trees can implement the *same* interface.



Abstract Class VS Interface

ABSTRACT CLASS	INTERFACE
Extends	Implements
Abstract method dan Concrete method	Pure abstract method
Single inheritance	Multiple implements (mirip dengan konsep multiple inheritance)

THE END