Westerdals

Oslo School of Arts,
Communication and Technology

PGR 101 Objektorientert Programmering 2
Vår 2017

Forelesning 21.3.17

(Stein Marthinsen – marste@westerdals.no)

Lag kryssordet!

Head First Java, 2nd Edition (Paperback)

by Kathy Sierra, Bert Bates

Dagens tema

Casting

Mer abstract/Interface

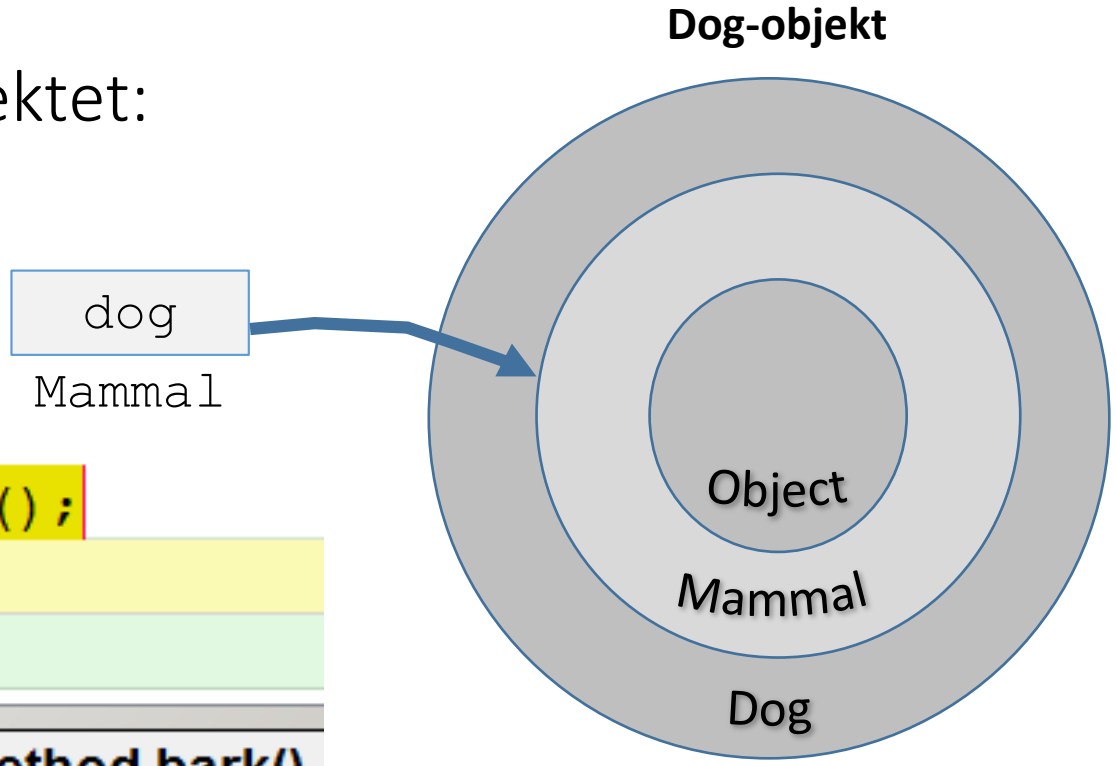
Casting

Fra forrige gang:

Hvis et objekt opprettes slik:

```
Mammal dog = new Dog();
```

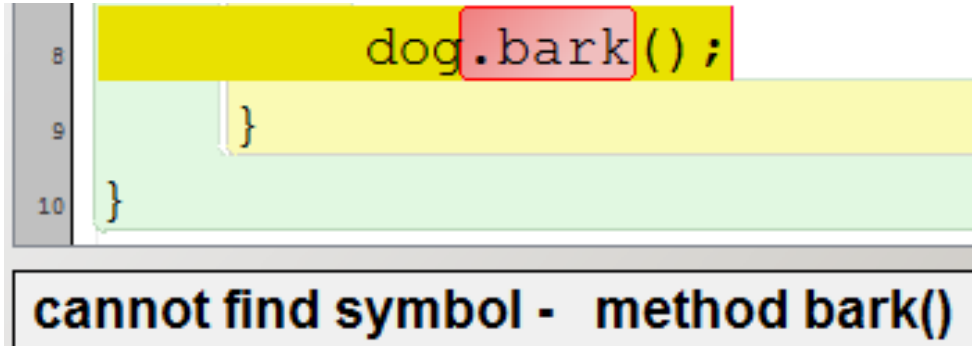
har vi bare tilgang til `Mammal`-delen av objektet:



Det vil altså si at

```
dog.bark();
```

vil gi feilmelding.



HUSK!

Hvis vi skal få tilgang til `Dog`-egenskapene, må vi "caste" (type-omforme) referansen til type `Dog` (som vi vet er der!)

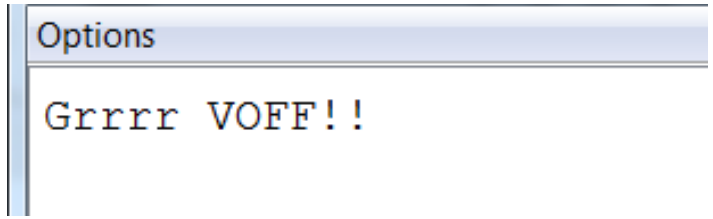
```
Dog aDog = (Dog) dog;
```

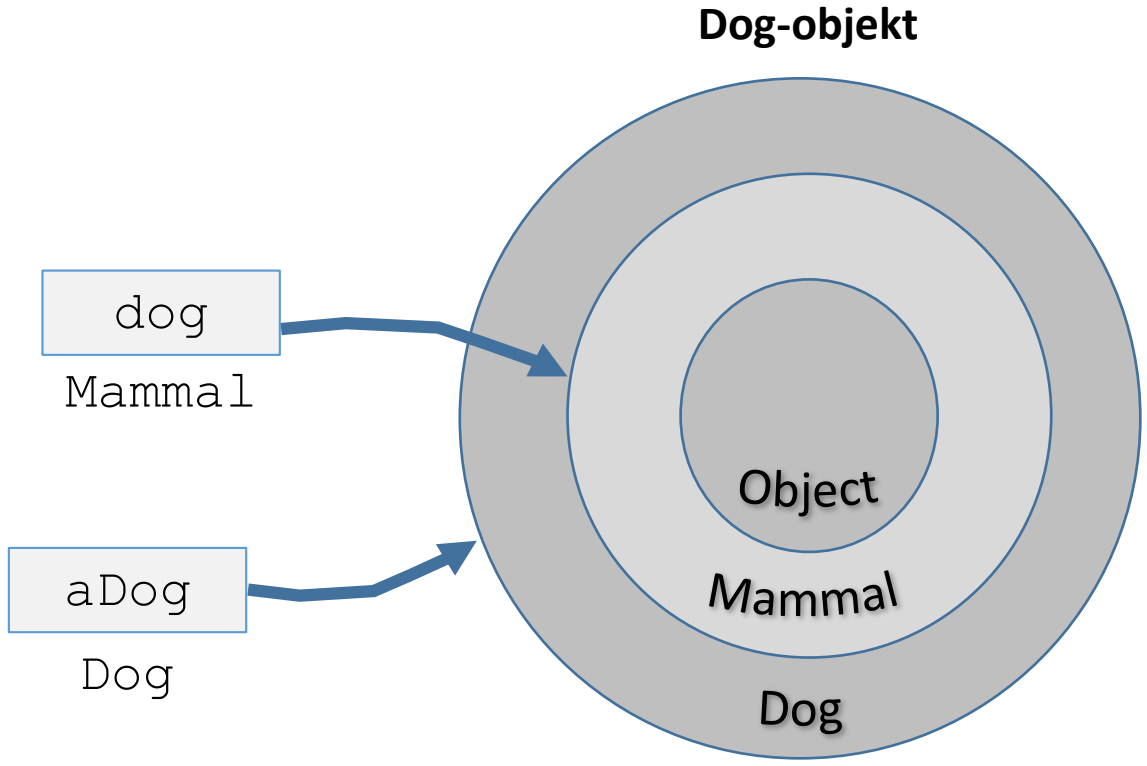
Nå har vi tilgang til alt i objektet!

Nå vil

```
aDog.bark();
```

fungere.



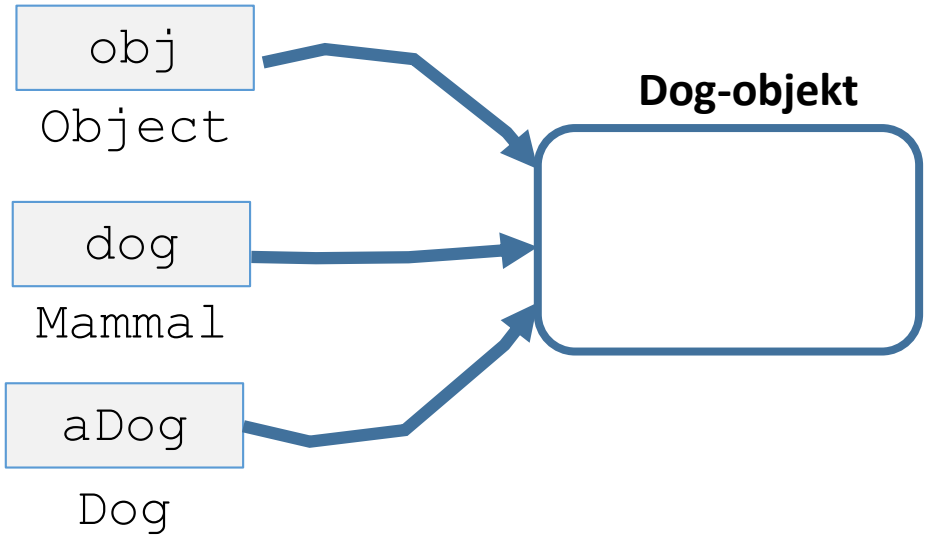


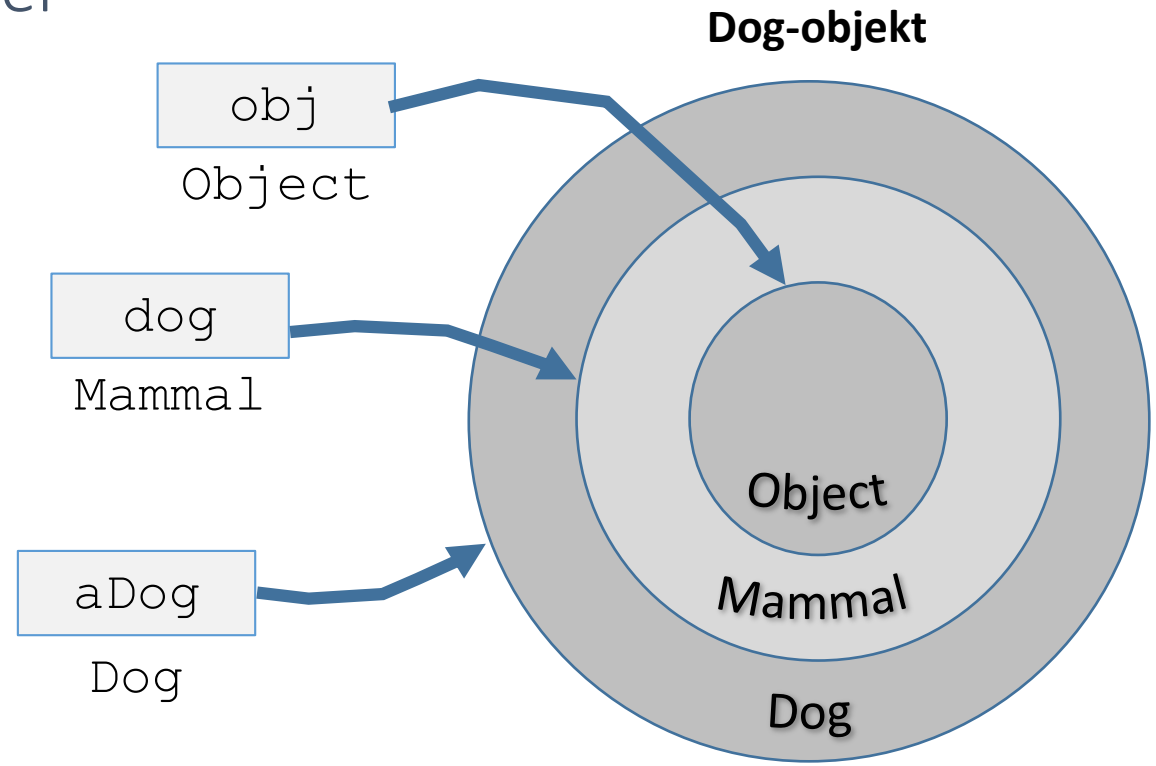
HUSK!

Med:

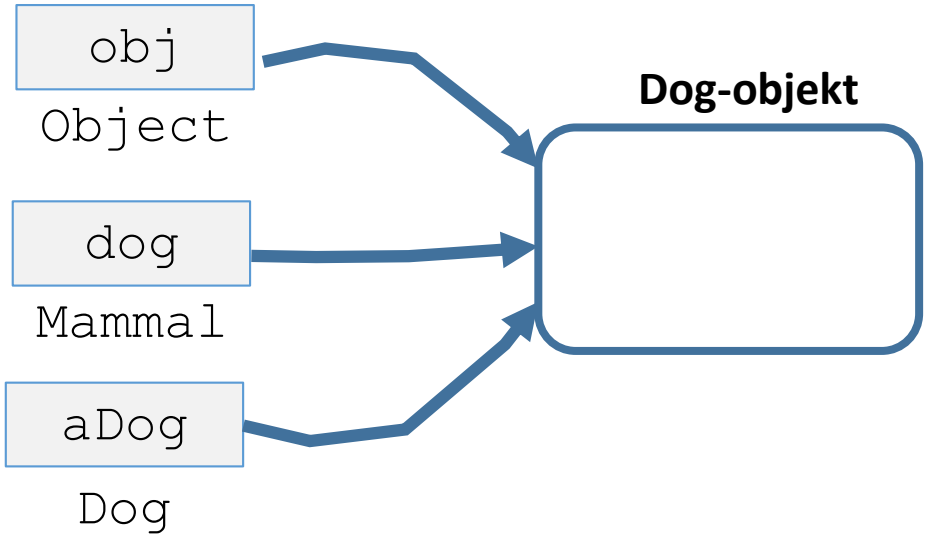
```
Object obj = aDog();
```

vil vi ha tilgang til bare `Object`-egenskaper
(selv om den refererer `Dog`-objektet!)

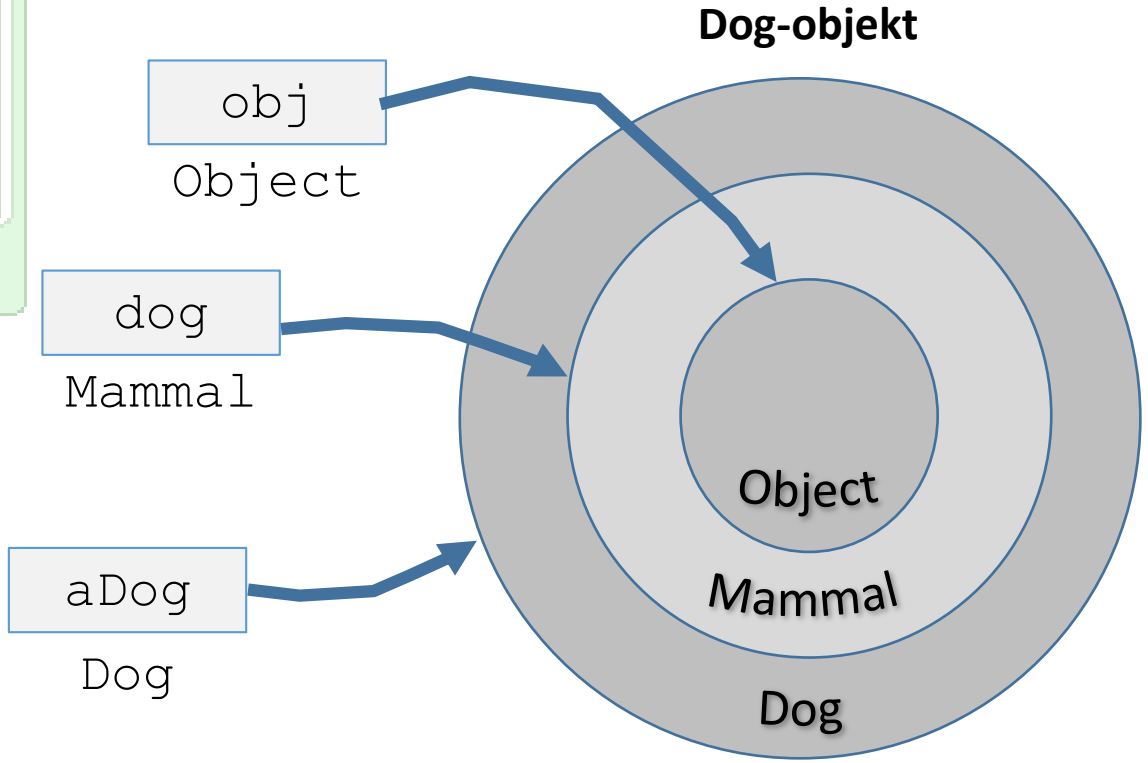




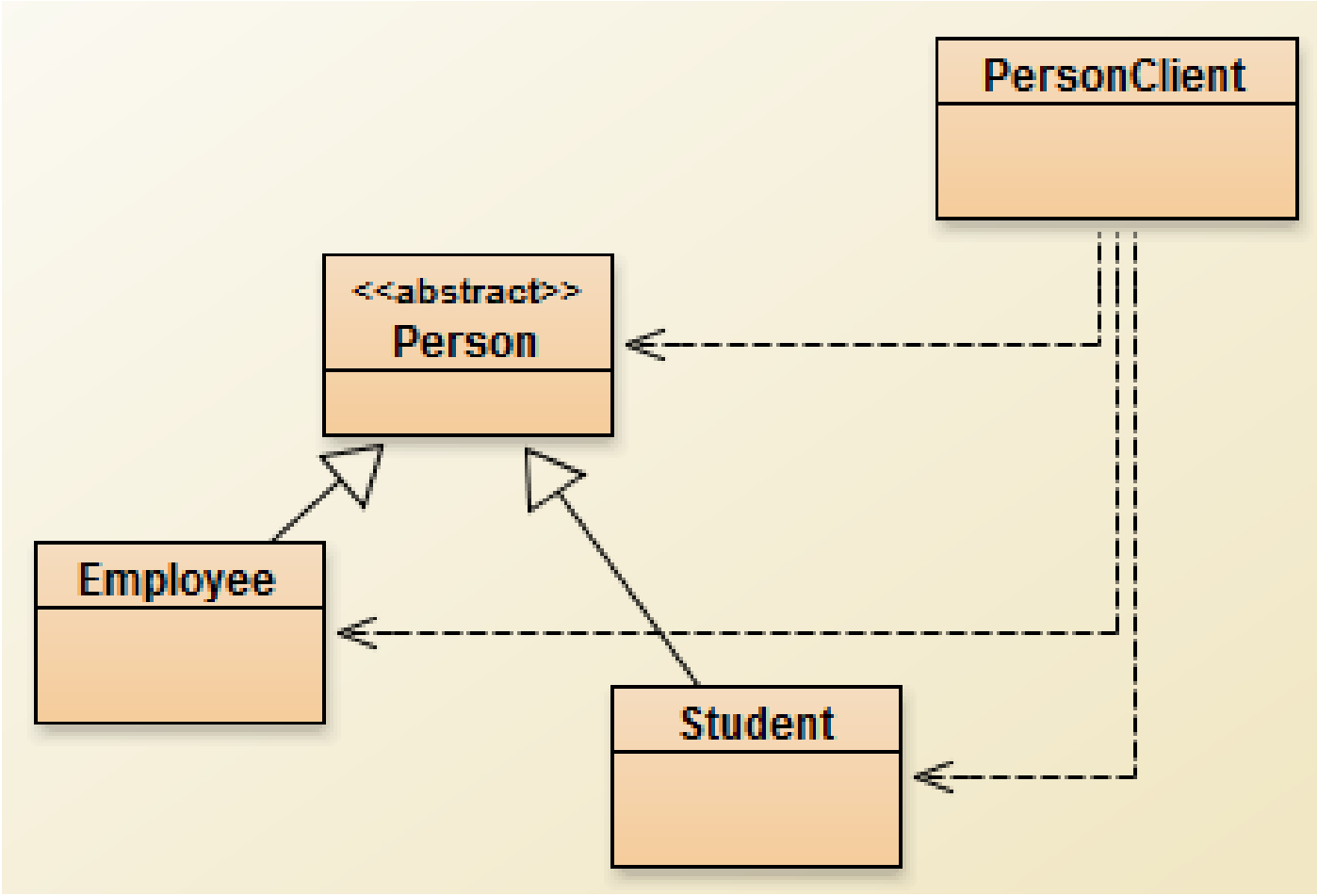
```java
public class MammalClient {
    public void method() {
        Dog aDog = new Dog();
        Object obj = aDog;
        Mammal dog = aDog;
        aDog.bark();
    }
}
```



Her vil vi altså ha tilgang til alle Dog-egenskaper bare med referansen aDog (selv om alle tre refererer samme objekt!)



En samling av objekter



Klassen `Person`

```
public abstract class Person implements Comparable <Person>{
    //Attributter
    private String firstName;
    private String familyName;
    private String socialSecurityNumber;

    public Person() {
        this("ukjent", "ukjent", "ukjent");
    }


    public Person(String firstName, String familyName, String socialSecurityNumber) {
        setFirstName(firstName);
        setFamilyName(familyName);
        setSocialSecurityNumber(socialSecurityNumber);
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
```

Collections.sort(...) bruker/forutsetter compareTo

```
import java.util.*;

public class PersonClient {
    public void mainMethod() {
        ArrayList<Person> persons = new ArrayList<Person>();
        persons.add(new Student("2345", "98769876", "Albin", "Albinsen", 20));
        persons.add(new Student("2312", "97939793", "Joabin", "Joabinsen", 25));
        persons.add(new Student("3122", "79867986", "Sylfest", "Sylfestsen", 20));
        persons.add(new Employee("12345", "98709870", "Dragan", "Dragansen", 250000));
        persons.add(new Employee("23451", "87698769", "Petrus", "Petrussen", 300000));
        persons.add(new Employee("34512", "76987698", "Doris", "Dorissen", 450000));

        Collections.sort(persons);

        for (Person p : persons) {
            System.out.println(p);
        }
    }
}
```

Klassen `Person`

1. *Implementerer* **`interface Comparable`**.

```
public abstract class Person implements Comparable <Person>
```

2. Har metoden **`compareTo`**.

```
public int compareTo(Person p) {
    int result = getSocialSecurityNumber().
            compareTo(p.getSocialSecurityNumber());
    return result;
}
```

`String` sin **`compareTo`**

interface

En klasse som skal brukes på en bestemt måte, må (kanskje) oppfylle bestemte krav.

Et interface kan oppfattes som formulering av slike krav.

Objekter av en klasse oppfyller slike krav ved at *klassen* **"implementerer"** et (eller flere) interface.

```
public abstract class Person implements Comparable <Person>
```

I dette tilfellet vil vi f.eks. kunne *sortere* en samling **`Person`**-objekter.

ArrayList.contains(...) bruker equals(...)

```
Student s = new Student("2312", "97939793", "Joabin", "Joabinsen", 25);

System.out.println("Er " + s + " i listen?");
System.out.println(persons.contains(s));
```

Options

```
Er (97939793) Joabin Joabinsen (2312) 25 i listen?
true
```

contains

```
public boolean contains(Object o)
```

Returns `true` if this list contains the specified element. More formally, returns `true` if and only if this list contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`.

ArrayList.contains(...) bruker equals(...)

For å kunne bruke denne metoden, må vi altså ha laget en tilpasset `equals`-metode i klassen "vår".

Gjør vi ikke det, vil den arvede versjonen fra `Object` bli brukt!

Og hvordan fungerer den????

ArrayList.contains(...) bruker equals(...)

```
Student s = new Student("2312", "97939793", "Joabin", "Joabinsen", 25);

System.out.println("Er " + s + " i listen?");
System.out.println(persons.contains(s));
```

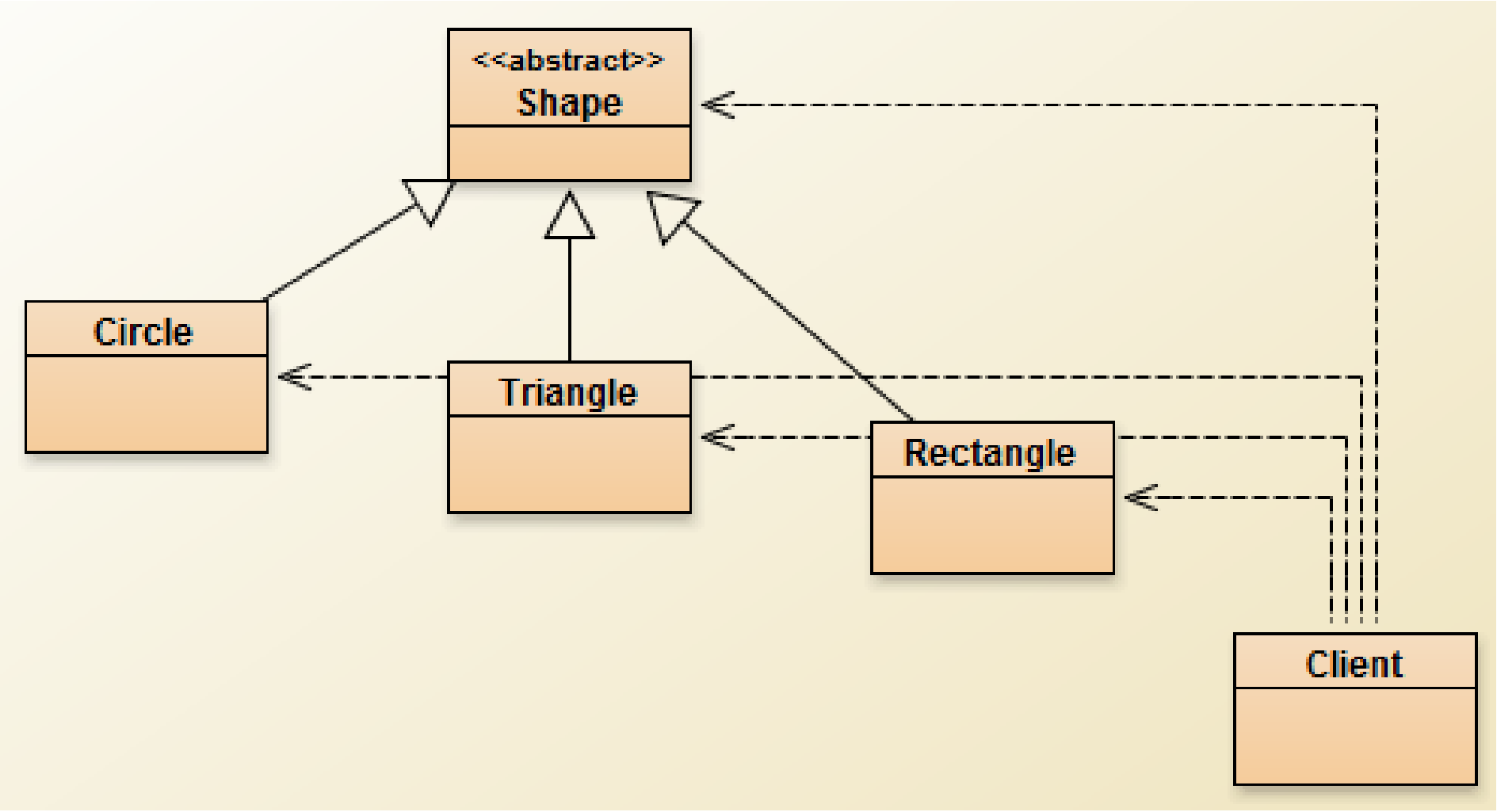
`equals`-metoden i klassen er her kommentert vekk!

Options

```
Er (97939793) Joabin Joabinsen (2312) 25 i listen?
false
```

```
ArrayList<Person> persons = new ArrayList<Person>();
persons.add(new Student("2345", "98769876", "Albin", "Albinsen", 20));
persons.add(new Student("2312", "97939793", "Joabin", "Joabinsen", 25));
persons.add(new Student("3122", "79867986", "Sylfest", "Sylfestsen", 20));
persons.add(new Employee("12345", "98709870", "Dragan", "Dragansen", 250000));
persons.add(new Employee("23451", "87698769", "Petrus", "Petrussen", 300000));
persons.add(new Employee("34512", "76987698", "Doris", "Dorissen", 450000));
```

Shape



Shape

Var deklarert som superklasse for klassene `Circle`, `Triangle` og `Rectangle`.

Metodene `getArea` og `getPerimeter` er deklarert `abstract` fordi `Shape` ikke "vet" hvordan areal og omkrets beregnes for vilkårlige figurer.

Hvorfor da legge disse i `Shape`???

```
public abstract class Shape {
    public Shape() {
    }

    abstract public double getArea();

    abstract public double getPerimeter();
}
```

Shape

For da kan vi...

```
import java.util.ArrayList;
public class Client {
    public void mainMethod() {
        ArrayList<Shape> shapes = new ArrayList<Shape>();
        shapes.add(new Rectangle(18, 18));
        shapes.add(new Triangle(30, 30, 30));
        shapes.add(new Circle(12));

        for (Shape s : shapes) {
            System.out.println("Areal = " + s.getArea() +
                "\nOmkrets = " + s.getPerimeter());
        }
    }
}
```

Fordi `Shape` er super-klasse!

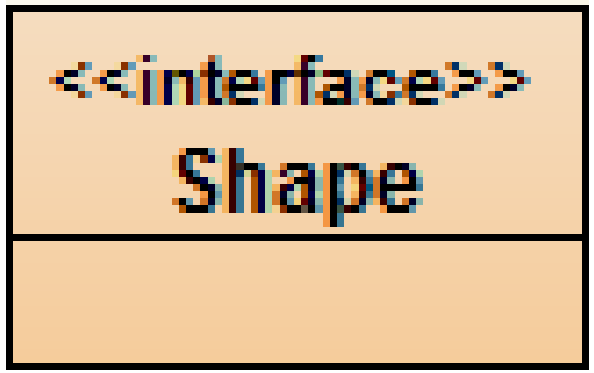
...og inneholder disse metodene!

Shape – slik den er nå

```
public abstract class Shape {
    public Shape() {
    }

    abstract public double getArea();

    abstract public double getPerimeter();
}
```

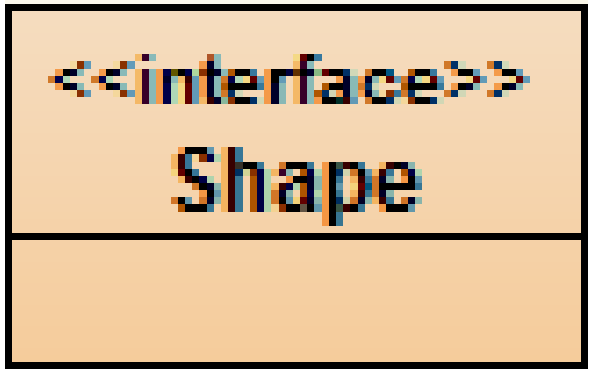
Alternativ Shape

```
public interface Shape {
    public double getArea();

    public double getPerimeter();
}
```



Alternativ Shape

```
public interface Shape {
    double getArea();

    double getPerimeter();
}
```



Alternativ Circle, Triangle og Rectangle

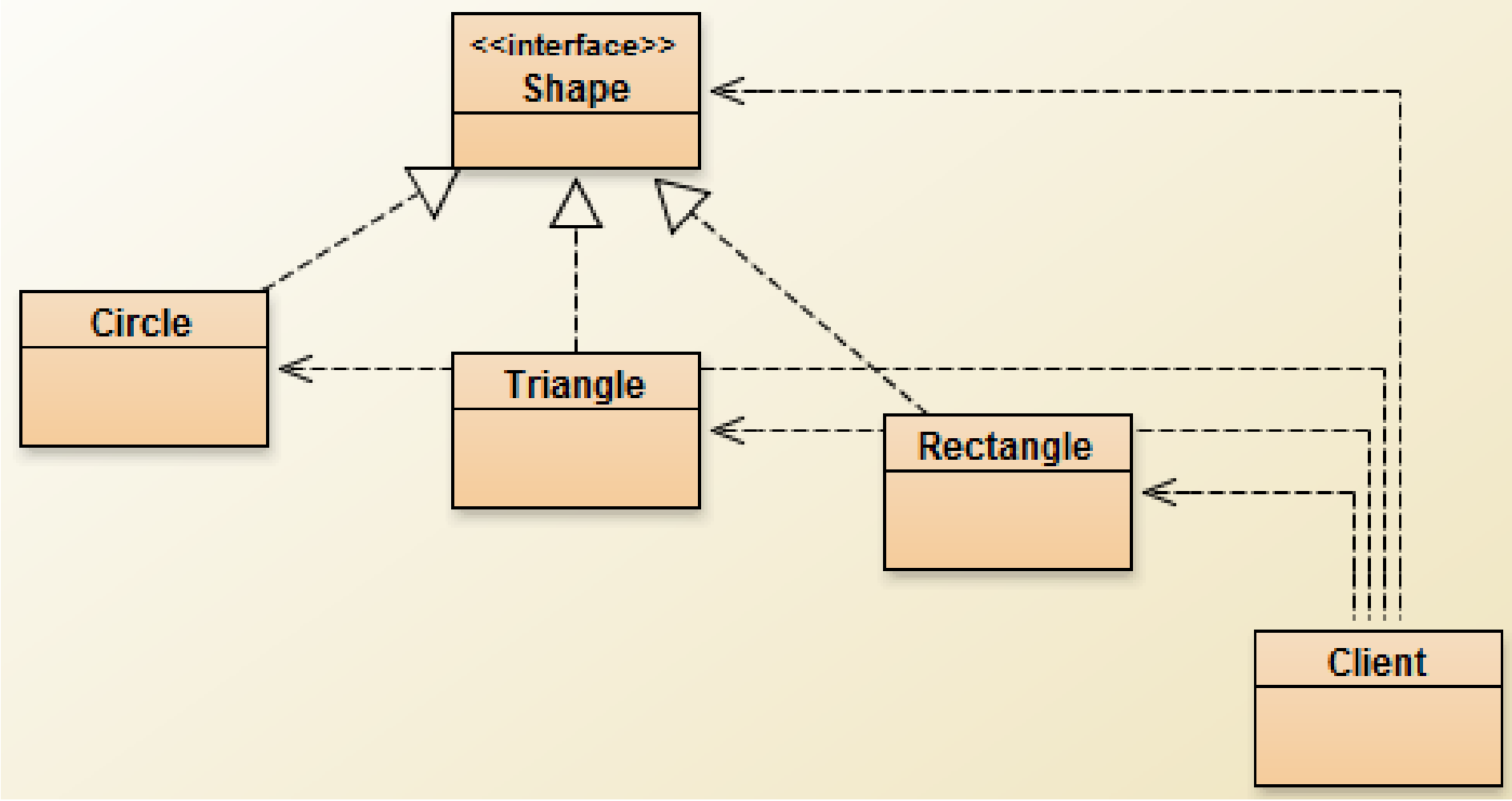
```
public class Circle implements Shape {
```

```
public class Triangle implements Shape {
```

```
public class Rectangle implements Shape {
```

Resten i disse klassene – og klassen Client – kan være som det er!

Alternativ Shape, Circle, Triangle og Rectangle



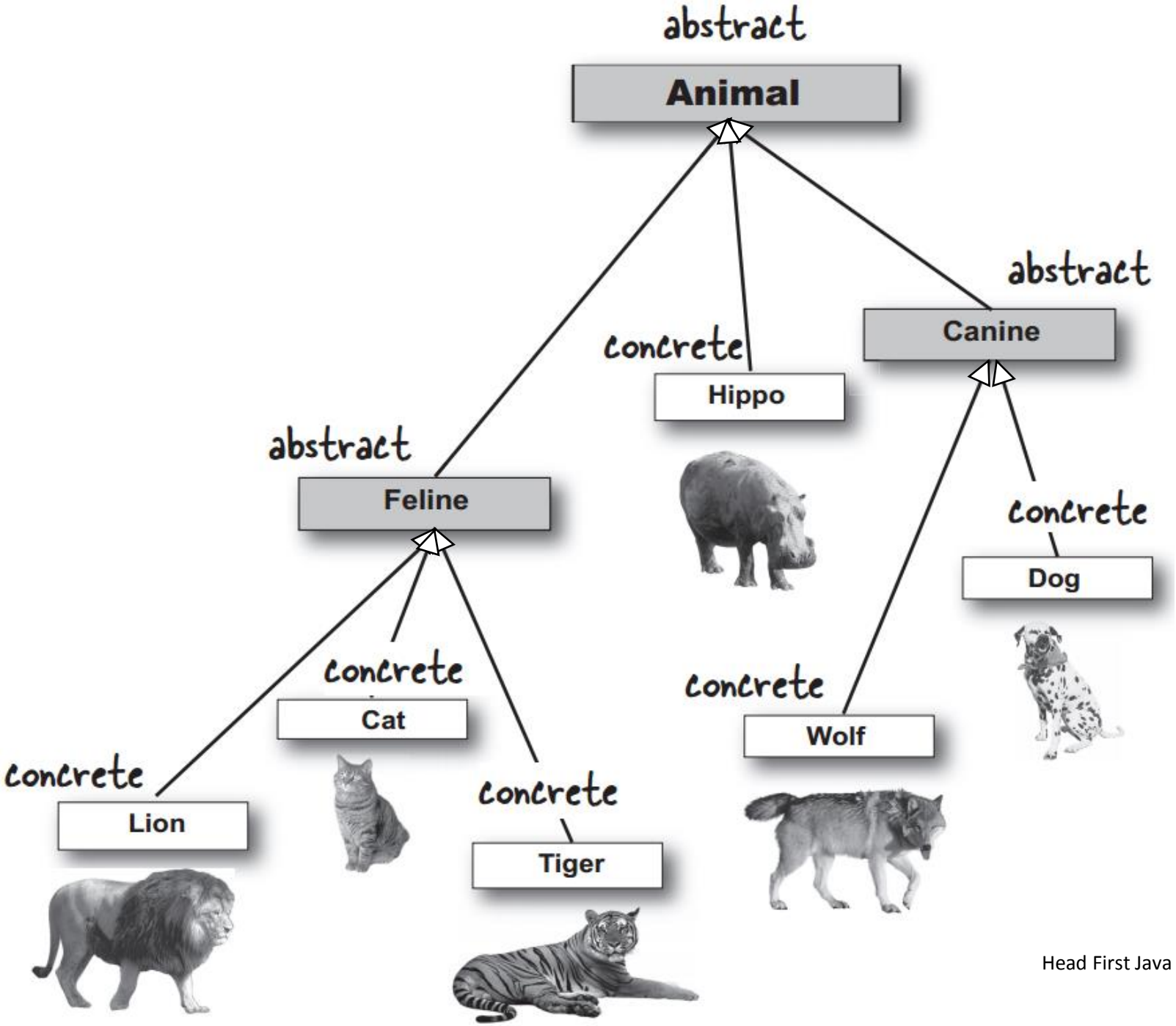
Interface

Med `Shape` som interface, *tvinges* klassene `Circle`, `Triangle` og `Rectangle` til å lage sine egne versjoner av metodene `getArea` og `getPerimeter`.

Hvis `Circle`, `Triangle` og `Rectangle` "vil være" en `Shape`, må de altså oppfylle kravet – å lage metodene `getArea` og `getPerimeter`.

Det er konsekvensen av å implementere interfacet.

Med interface kan vi altså "bestemme" hvordan sub-klasser skal være!

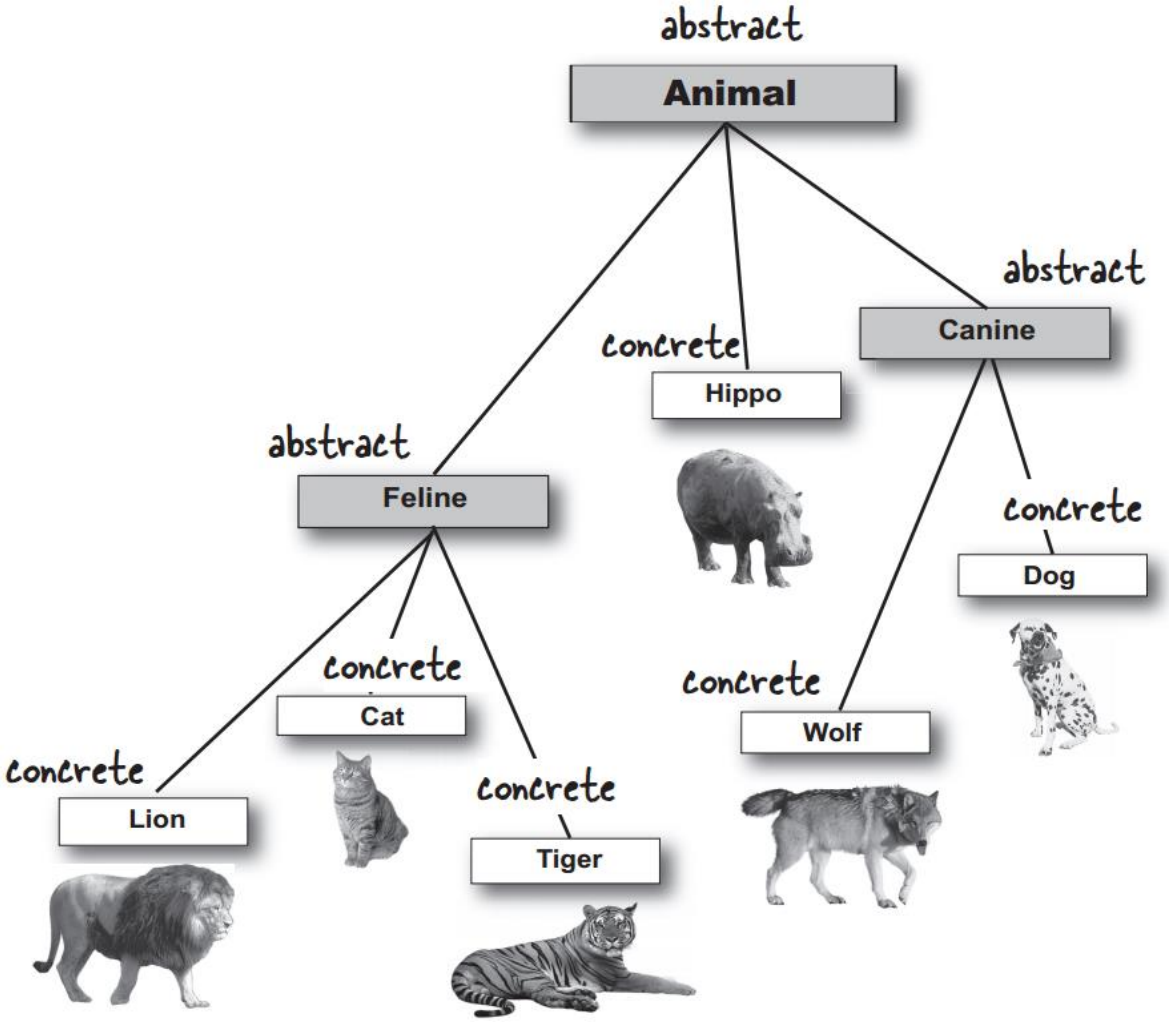


Head First Java Your Brain on Java - A Learner's Guide

Utfordring

I figuren har vi dyr som også kan betraktes som kjæledyr – med kjæledyr-egenskaper (kan koses med, klappes, lekes med osv.)

Hvordan løser vi dette?



Alternativ 1:

Legger disse egenskapene inn i superklassen `Animal`, men gjør dem *`abstract`*

Fordel:

tvinger alle subklassene til å implementere dem – de som ikke vil være kjæledyr kan lage tomme metoder

Ulempe:

Alle subklassene MÅ implementere dem – hvis ikke blir de selv abstrakte

Dermed vil de *utad også* se ut som om de er kjæledyr – har kjæledyregenskaper

Alternativ 2:

Legger de aktuelle egenskapene inn i de aktuelle kjæledyrklassene (`Cat`, `Dog`, ...)

Fordel:

kjæledyregenskapene ligger der de hører hjemme

Ulempe:

Må kanskje cast'e referanser for å få tak i disse egenskapene

Alternativ 3:

Kan man ikke lage *to* superklasser?

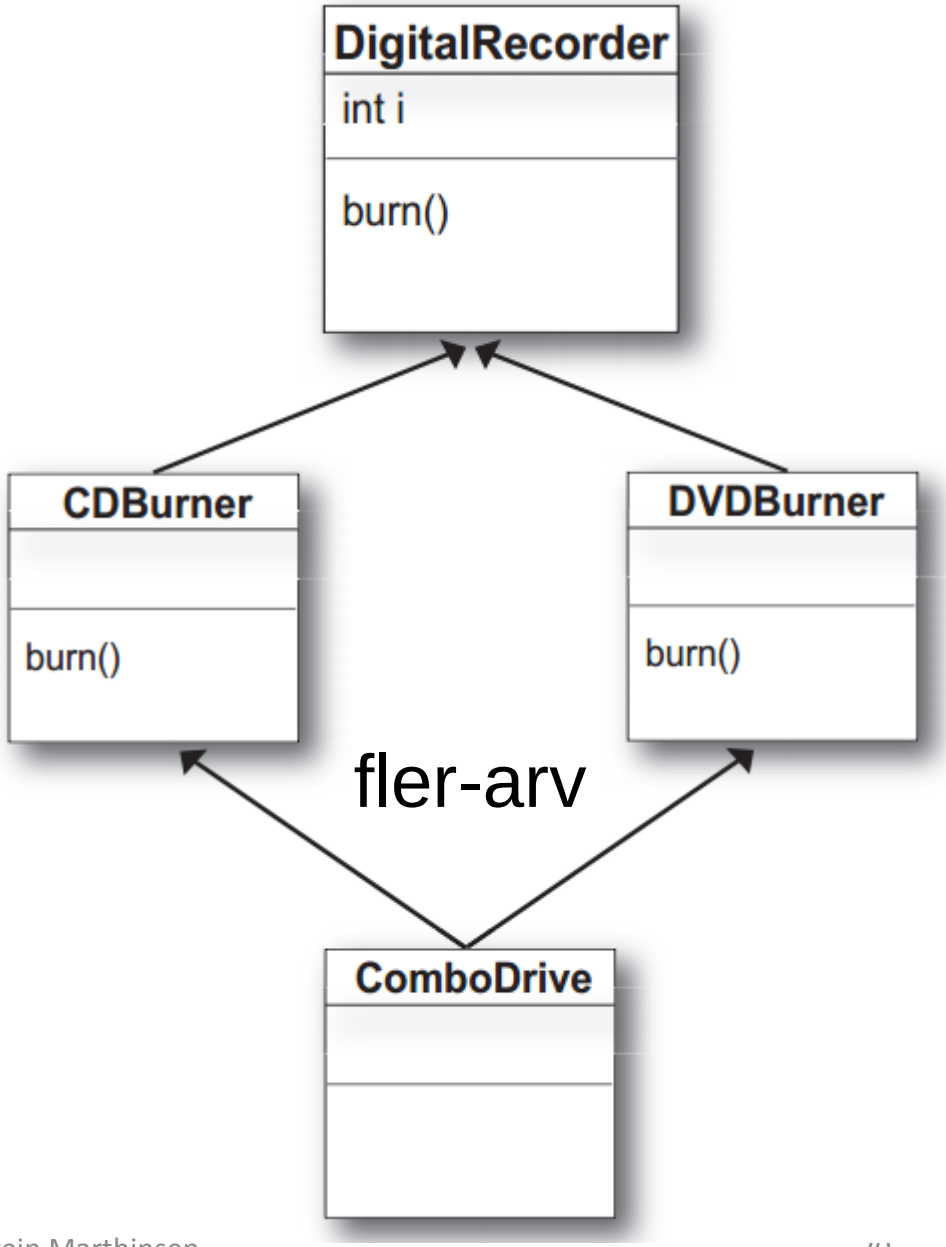
```
public class Dog extends Animal, Pet {
  //...
}

public class Cat extends Animal, Pet {
  //...
}
```

NEI – det går ikke i Java!

Tenk om...

```
ComboDrive combo
   = new ComboDrive(...);

combo.burn();
```

Hvilken burn-metode?

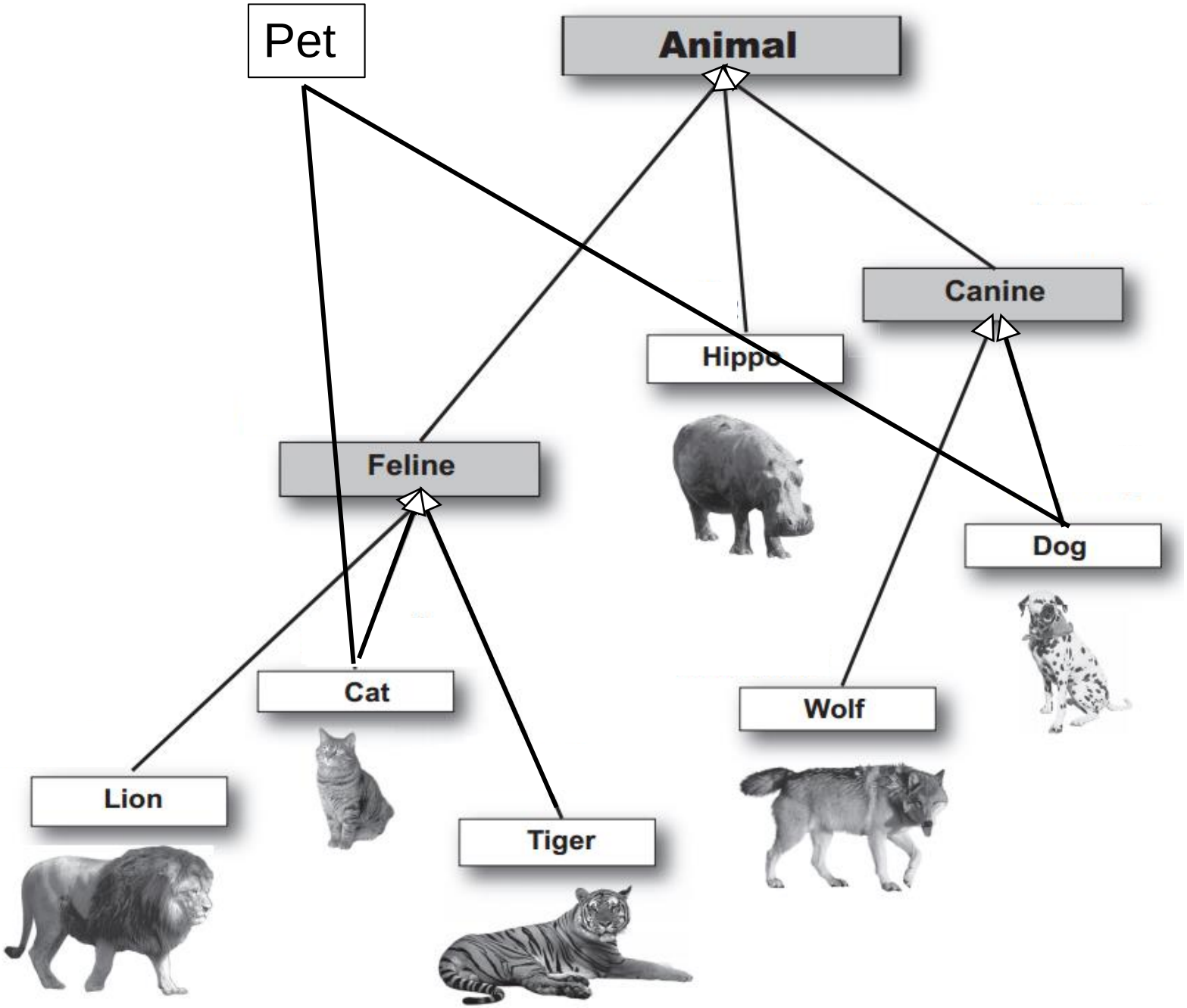


Løsning

```
public interface Pet {
    public abstract void beFriendly();
    public abstract void play();
    ... //andre Pet-egenskaper
}

public class Dog extends Animal implements Pet {
    //må implementeres
    public void beFriendly() {...}
    public void play() {...}

    //andre hunde-egenskaper
    public void roam {...}
    public void sit() {...}
    public void bark() {...}
}
```

En hund er etter dette et dyr med kjæledyregenskaper!

interface



Med `interface`

Bare kjæledyrklassene får (må ha) kjæledyregenskaper.

Kan nå bruke en felles `Pet`-referansetype:

Slik:

```
ArrayList<Pet> pets = new ArrayList<Pet>();
pets.add(new Cat(...));
pets.add(new Dog(...));
...
```

Eller slik:

```
ArrayList<Animal> animals = new ArrayList<Animal>();
animals.add(new Cat(...));
animals.add(new Hippo(...));
animals.add(new Tiger(...));
...
```

Oppgaver på øvingen

Skriv skisser for klasser/interface i eksemplet Animal/Pet