NTNU

Kunnskap for en bedre verden

Øvingsforelesning 5 (del 1/2)

TDT4100 Objektorientert programmering

22.02.2023

Jostein Hjortland Tysse
Vitenskapelig assistent
jostein.h.tysse@ntnu.no

NTNU

Interface / grensesnitt

Introduksjon og teori

Begrepsavklaring

- I dag skal vi kun prate om det som heter **grensesnitt og interfaces** (og disse to begrepene er de samme, bare på norsk og engelsk) og **ikke** brukergrensesnitt
- Jeg kommer til å bruke begrepet **interface** om det som vi gjennomgår i dag for å unngå potensiell forvirring
- **Brukergrensesnitt** er et helt separat begrep og omhandler grafiske elementer (eks. JavaFX), og har lite med det vi snakker om i dag å gjøre

Interface: litt repetisjon

- Et interface er en referansetype i Java som består av en samling av **abstrakte metoder**. Klasser kan **implementere** interface, og derav **arve** de abstrakte metodene til interfacet.

- En **abstrakt metode** er en metode uten kropp, som vil si at den ikke er implementert ennå. Merk hvordan metoden nedenfor slutter med semikolon:

```
public void metodeSomErFellesForAlleUndertyper();
```

- Dette er nyttig fordi det lar oss definere en **struktur** / **et sett med regler** for klassene som implementerer interfacet. Alle klassene som implementerer interfacet må **også implementere interfacets metoder**.

Interface vs. klasser

Et **interface** har en del fellestrekk med en klasse, men det er også en del svært **viktige forskjeller**:

- Et interface kan **ikke** ha felter (med unntak av *statiske* felter)
- Alle metodene i et interface er **abstrakte** (ingen innhold)
- Alle metoder definert i et interface er automatisk **public**

  Vi har **ingen konstruktør** i interface
- **Alle** felter (variabler) som defineres i et interface er **automatisk**

  `public static final`

Forskjellige typer interfaces

- Det finnes allerede **forhåndsdefinerte interfaces** i Java, som for eksempel **Comparable** og **Iterable**. Dette er nyttig når man vil lage klasser som skal ha en spesifikk funksjonalitet, som i dette tilfellet er å sammenligne eller iterere gjennom objekter.

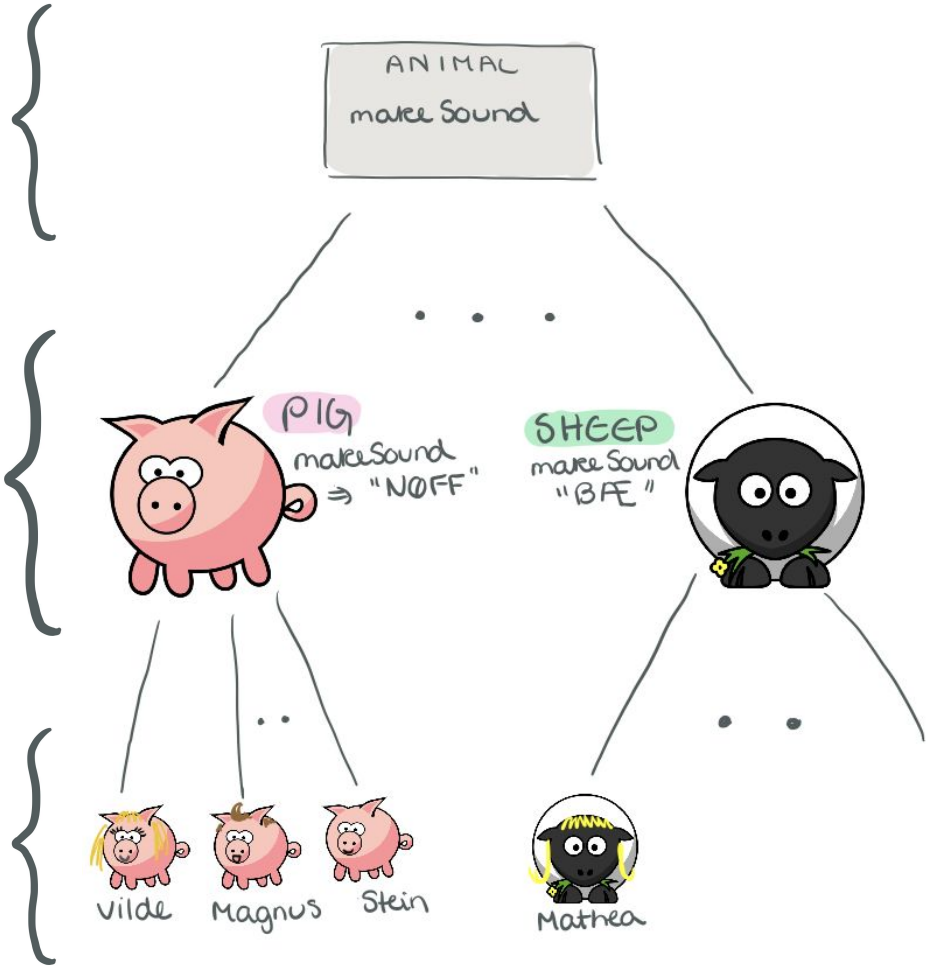
- Man kan også **lage sine egne interface** dersom man har lyst til å spesifisere en **struktur for funksjonaliteten** til en eller flere klasser på egenhånd.

- I dag og neste uke skal vi se på eksempler av begge deler.

Animal-interfacet sier at et objekt som **implementerer** Animal **må** ha en makeSound-metode som returnerer en streng. Dette gir en viss struktur (og forutsigbarhet) til objekter som implementerer Animal.

Klassene Pig og **Sheep** implementerer interfacet og må derfor også definere en makeSound()-funksjon. Merk at denne ikke er lik for de to klassene.

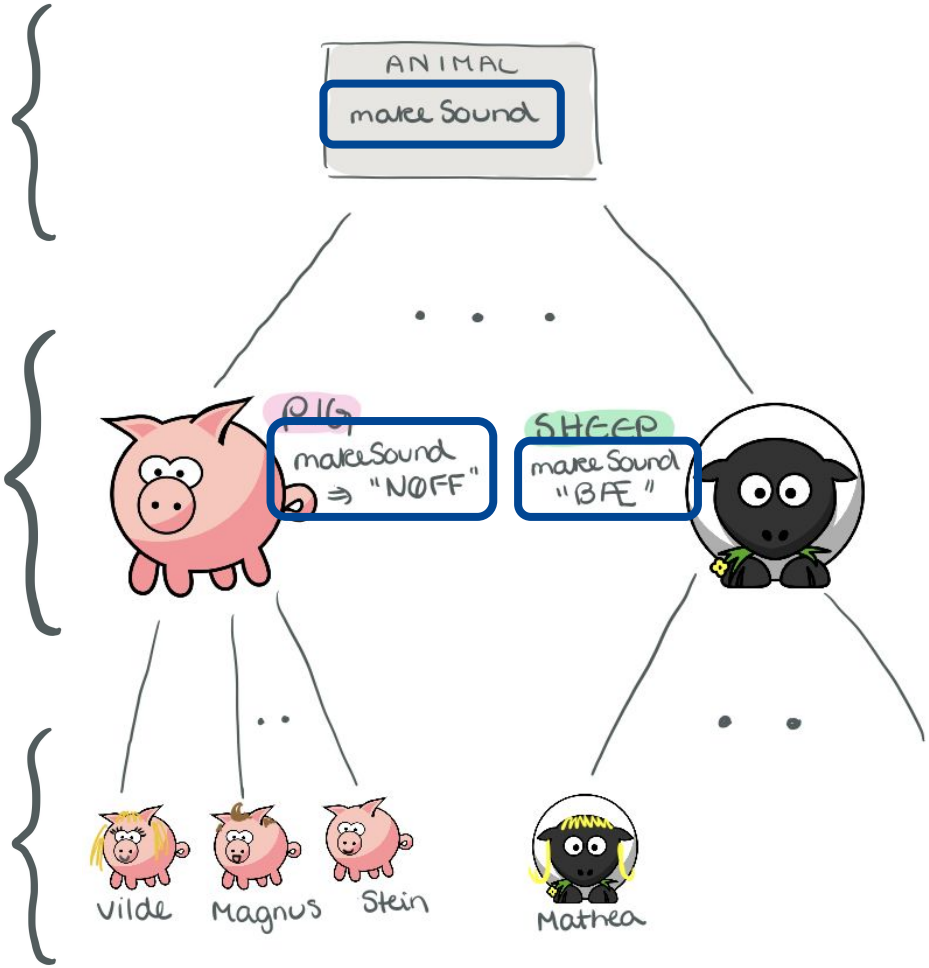
Instanser av klassene er som dere er kjent med tidligere, spesifikke objekter som opprettes vha. New.



Animal-interfacet sier at et objekt som **implementerer** Animal **må** ha en makeSound-metode som returnerer en streng. Dette gir en viss struktur (og forutsigbarhet) til objekter som implementerer Animal.

Klassene Pig og **Sheep** implementerer interfacet og må derfor også definere en makeSound()-funksjon. Merk at denne ikke er lik for de to klassene.

Instanser av klassene er som dere er kjent med tidligere, spesifikke objekter som opprettes vha. New.



Interface

- I Java er et interface en **type** som deklarerer metoder, men ikke tilbyr implementasjon av metodene

- Det reserverte nøkkelordet **implements** brukes for å indikere at en klasse implementerer et interface

- Hva sier et interface om klassene som implementerer de?
  - *"Dette er **metodene** jeg tilbyr som er **public***

Eksempel

Vi ekstraherer et interface fra Person-klassen

```java
public class Person {

  private String name;
  private int age;


  public Person(String name, int age) {
      this.name = name;
      this.age = age;
  }

  public String getName() {
      return this.name;
  }

  public void setName(String name) {
      this.name = name;
  }
}
```

Eksempel

Vi ekstraherer et interface fra Person-klassen

```java
public class Person {

  private String name;
  private int age;


  public Person(String name, int age) {
      this.name = name;
      this.age = age;
  }

  public String getName() {
      return this.name;
  }

  public void setName(String name) {
      this.name = name;
  }
}
```

Eksempel

Vi ekstraherer et interface fra Person-klassen

```java
public interface NamedObject {

  public String getName();

  public void setName(String name);

}
```

```java
public class Person {

  private String name;
  private int age;


  public Person(String name, int age) {
      this.name = name;
      this.age = age;
  }

  public String getName() {
      return this.name;
  }

  public void setName(String name) {
      this.name = name;
  }
}
```

Eksempel

Vi ekstraherer et interface fra Person-klassen

```java
public interface NamedObject {

  public String getName();

  public void setName(String name);

}
```

```java
public class Person implements NamedObject {

  private String name;
  private int age;


  public Person(String name, int age) {
      this.name = name;
      this.age = age;
  }

  public String getName() {
      return this.name;
  }

  public void setName(String name) {
      this.name = name;
  }
}
```

Eksempel

Vi ekstraherer et interface fra Person-klassen

```java
public interface NamedObject {

  public String getName();

  public void setName(String name);

}
```

```java
public class Person implements NamedObject {

  private String name;
  private int age;


  public Person(String name, int age) {
      this.name = name;
      this.age = age;
  }

  public String getName() {
      return this.name;
  }

  public void setName(String name) {
      this.name = name;
  }
}
```

Noen viktige moment

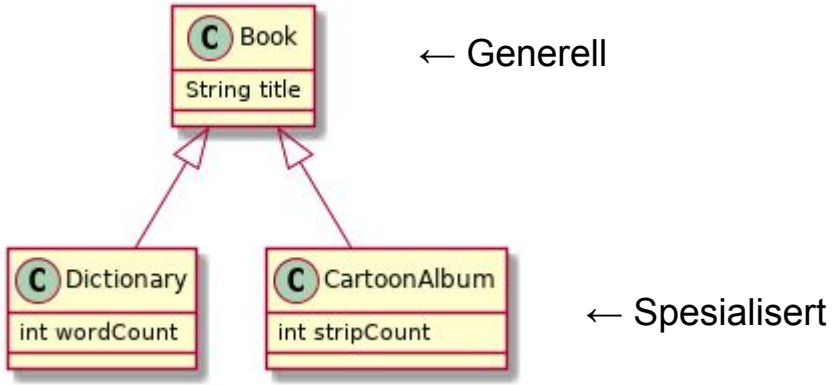
- Merk at alle metodene er “tomme” (abstrakte)
- Vi har heller ikke definert noen felter her, men vi kunne lagt ved noen **statiske** felter her (som også må være **public** og **final**)
- Alle metodene definert i et interface må defineres når en klasse implementerer interfacet

```
public interface NamedObject {

  public String getName();

  public void setName(String name);

}
```

Arv

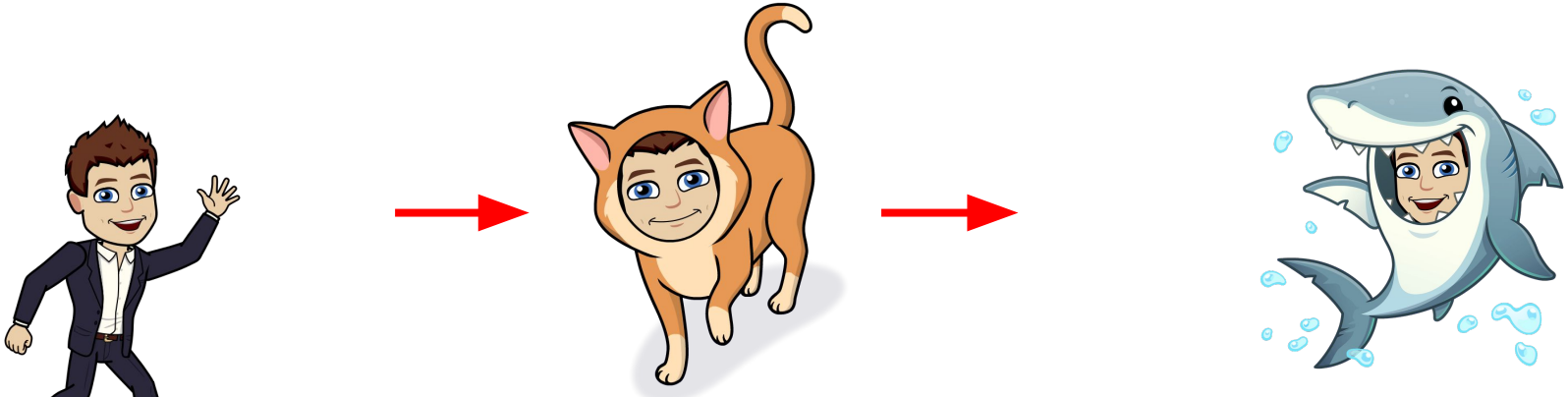
- Objekter av forskjellige typer i java struktureres i et arvingshierarki, som går fra mest generell (øverst) til mest spesifikk (nederst)

- En klasse kan også arve metoder fra / implementere ett eller flere interface



Polymorfisme

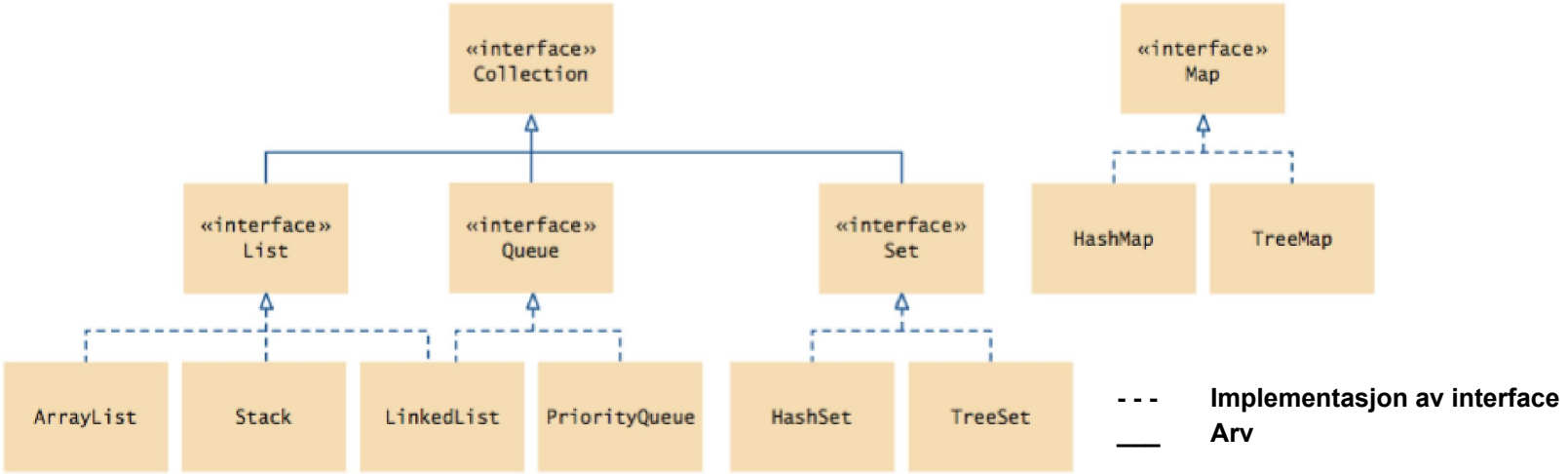
- *Det å velge en metode blant flere metoder med samme navn, på grunnlag av de faktiske typene til de implisitte parametrene*
- Et objekt kan ta **flere mulige former**, et **`Dog`**-objekt kan ta form som et objekt av typen **`Dog`** men også av typen **`Animal`**
- interface muliggjør polymorfisme; dvs. mulighet til å behandle objekter med forskjellig oppførsel på en uniform måte (siden de har like metoder)



Collection-rammeverket

- Javas Collection-rammeverk er også et hierarki av interface-typer og -klasser
  - Lar deg implementere mange **forskjellige** måter å lagre, organisere og bearbeide grupper av objekter

- Interfacene og klassene ligger i følgende hierarki:



Øving 5

Interface (og litt funksjonell programmering)

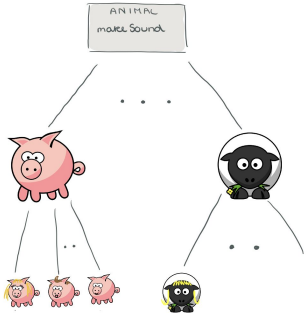
Oppgaveløsning

Vi lager en bondegård med dyr :)

Oppgave 1

Vi kommer til å ha forskjellige typer (klasser) dyr, og ønsker å ha et felles sett med metoder tilgjengelig for dyrene.

Lag et interface Animal som har tre metoder: *getName* og *getAge* som fungerer som get-metoder, og *makeSound* som gir ut en tekststreng som representerer en dyrelyd



Oppgave 2

Vi har nå laget et interface som definerer strukturen til et dyr, men vi har ikke definert noen dyr ennå.

Lag to klasser som implementerer Animal-interfacet. Vi kommer til å lage Dog og Chicken ved gjennomgang, men du kan fint lage andre dyr hvis du ønsker det.

Oppgave 3

Vi har nå et lite sett med dyr, og kan begynne representasjonen av en gård.

Lag klassen *Farm* som tar vare på et sett med dyr. Klassen skal ha to metoder *addAnimal* og *getAnimals* som respektivt legger til dyr i en liste over alle dyrene på gården, og returnerer listen.

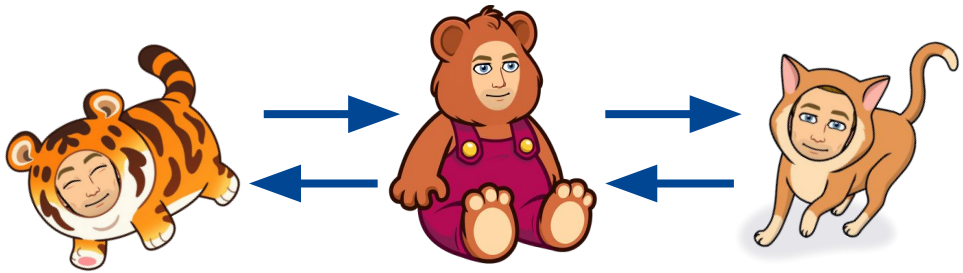
Comparable og Comparator

Innebygde interfaces for sammenligning i java

Comparable og Comparator

- Dette er to interface som brukes for å sammenligne objekter av samme type, spesielt i sammenheng med **sorteringsmetoder** i Java.

- Eksempel:
  - Vi ønsker å sortere **`Animal`**-objekter i en liste basert på alderen deres.



Comparable og Comparator

Vi ønsker å sortere flere **`Animal`**-objekter i en liste basert på alderen deres.

- Sortering kan vi gjøre ved å kalle den innebygde metoden **sort()**-på en liste.

Men, da må vi ta stilling til følgende:

- Hvordan vet Java hvilket attributt i **`Animal`** vi ønsker å sortere på? Skal vi sortere basert på navn, alder, eller noe annet?

I stedet for å bare gjette helt vilt så spesifiserer vi det heller **eksplisitt**, enten ved å **instansiere** et **`Comparator`**-objekt eller ved å **implementere** **`Comparable`**-interfacet.

Klassedefinisjonen:

```
public class Animal {
    private int age;
    private double weight;
    private String name;
}
```

I main-metode:

```
List<Animal> animals;
animals.sort();
```

Comparable og Comparator (2)

- Compar**ator**:
  - Objekt som lar oss sjekke om **et objekt** av type **T** er **mindre, lik eller større** enn **et annet objekt** av type **T**
  - *I tillegg: et funksjonelt interface*

- Compar**able**:
  - Interface som lar oss spørre om **dette** objektet (`this`), av type **T**, er mindre, lik eller større enn **et annet objekt** av type **T**

Comparator-interfacet

Eksempel

Implementeres som et **eksternt** objekt som evaluerer objektene vi ønsker å sammenligne

```
public class AnimalComparator implements Comparator<Animal>{
    @Override
    public int compare(Animal o1, Animal o2) {
        /* Metoden skal returnere et negativt tall hvis o1 < o1,
           0 hvis o1 = o2 og et positivt tall hvis o1 > o2 */
        return /* evalueringsfunksjon her */;
    }
}
```

```
AnimalComparator comparator = new AnimalComparator();
Collections.sort(animals, comparator);
```

Comparable-interfacet

Eksempel

Et alternativ til **`Comparator`**. I stedet for å opprette et separat objekt så lar vi heller objektene som skal sammenlignes/sorteres implementere **`Comparable`**-interfacet

```
public class Dog implements Animal, Comparable<Animal>{
    @Override
    public int compareTo(Animal o) {
        /* Metoden skal returnere et negativt tall hvis this < o,
           0 hvis this = o og et positivt tall hvis this > o */
        return /* evalueringsfunksjon her */;
    }
}
```

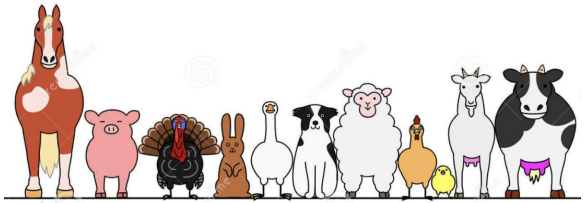
```
Collections.sort(dogs);
```

Merk at vi nå ikke trenger en egen **`Comparator`** for å sortere **`Dog`**-objekter

Oppgave 4

Vi har nå laget funksjonalitet for å representere en gård med dyr. Nå ønsker vi en funksjon å sortere dyrene på for at bonden skal kunne bestemme hvem han skal stelle først.

Lag en klasse AnimalAgeComparator som implementerer Comparator-interfacet. Denne skal sammenligne dyr basert på alderen deres.



instanceof

- Operatoren **`instanceof`** sjekker om et objekt(referanse) er en (under)type av en gitt type og returnerer true / false

- I begge eksemplene under vil vi få skrevet ut **`true`** fordi **`dog`** vil være en *instans av* både **`Pet`** og **`Dog`**

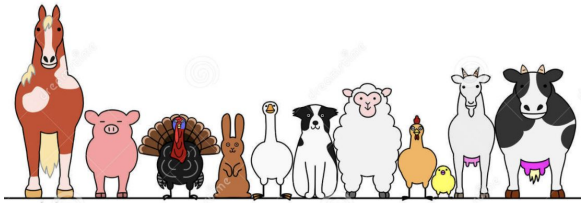
```
Pet dog = new Dog("Pluto");
System.out.println(dog instanceof Pet);
```

```
Pet dog = new Dog("Pluto");
System.out.println(dog instanceof Dog);
```

Oppgave 5

Bonden på gården ønsker å stille dyrene sine i en spesifikk rekkefølge. Vi skal derfor lage en funksjon å sortere dyrene basert på hvilken **type** de er.

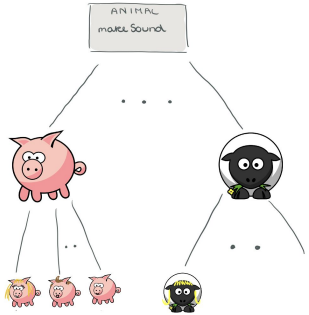
Lag en klasse AnimalTypeComparator som implementerer Comparator-interfacet. Dog-objekter skal stå først og Chicken-objekter skal plasseres bakerst.



Oppgave 6

Det kan være litt tungvint å måtte opprette et **Comparator**-objekt hver gang vi ønsker å sortere dyr, spesielt hvis vi ofte vil sammenligne på lik måte fra gang til gang. Vi kan i stedet definere en “standardsortering” for Animal-objekter vha. **Comparable**-grensesnittet.

La Animal-interfacet utvide Comparable-interfacet. Implementer nødvendige endringer i Dog- og Chicken-klassene. Vi kan basere oss på at standardsorteringen sorterer på alder.



Iterator og Iterable

Interfaces for iterasjon i java

Iterable

- Definerer at dette objektet er en type som kan *itereres over*, for eksempel med en **foreach**-løkke
- Vi vet allerede om en del ting som kan itereres over:
  - **`Collection, List, Map`**
- Ved å implementere **`Iterable`**-interfacet sier vi at klassen som implementerer det kan itereres over på samme måte som f.eks. **`List`**
- Når vi implementerer **`Iterable`**-interfacet må vi implementere følgende metode:

```
@Override
public Iterator<Dog> iterator() {
    return /* Et iterator-objekt for denne typen*/;
}
```

Iterable-interfacet

Eksempel

- Vi har en klasse som inneholder en liste med hunder:

```
public class Kennel implements Iterable<Dog> {
	List<Dog> dogs;
	public Kennel(Dog[] dogList) {
		this.dogs = new ArrayList<>(Arrays.asList(dogList));
	}
	@Override
	public Iterator<Dog> iterator() {
		return /* Et iterator-objekt for denne typen*/;
	}
}
```

- Kan da iterere over **Dog**-objekter i objekter av typen **Kennel** slik:

```
for(Dog dog : kennel) {
	dog.bark();
}
```

Typiske bruksscenarioer:

- **`Comparator`** og **`Comparable`**:
  - Du har en liste med *egendefinerte* objekter av en type og ønsker å kunne enkelt sortere disse i en liste.
- **`Iterable`**:
  - Du har en klasse som inneholder en liste av andre objekter og ønsker å iterere over disse
- **`Iterator`**:
  - Du ønsker å iterere over en liste eller lignende på en egendefinert måte, eller ønsker å fjerne elementer fra en liste *mens du itererer over den*
  - Se oppgavene **`CardContainer`**, **`StringGrid`** og **`StringMergingIterator`** i øving 6 for eksempler på dette

Oppgave 7

Vi ønsker nå å kunne iterere over alle dyrene i en gård. Dette kunne vi vanligvis gjort enkelt nok med bare å returnere en liste med objekt og bruke en for-each løkke på denne, men i stedet ønsker vi å abstrahere bort dette ved å implementere **Iterable**-grensesnittet.

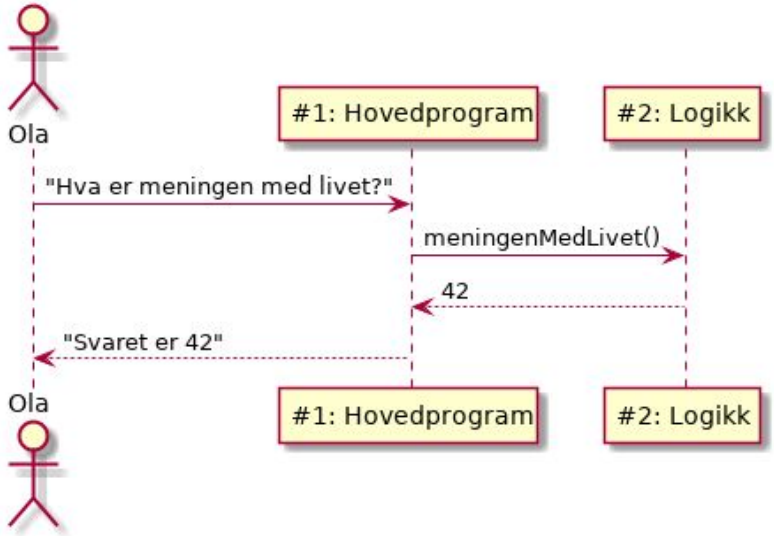
Implementer interfacet Iterable i Farm-klassen. Metoden vi må implementere skal la oss iterere over alle Animal-objekter i Farm

Sekvensdiagram

- Sekvensdiagrammer viser hvordan objekter samhandler, dvs. i hvilken sekvens objekter utfører metodekall på hverandre.

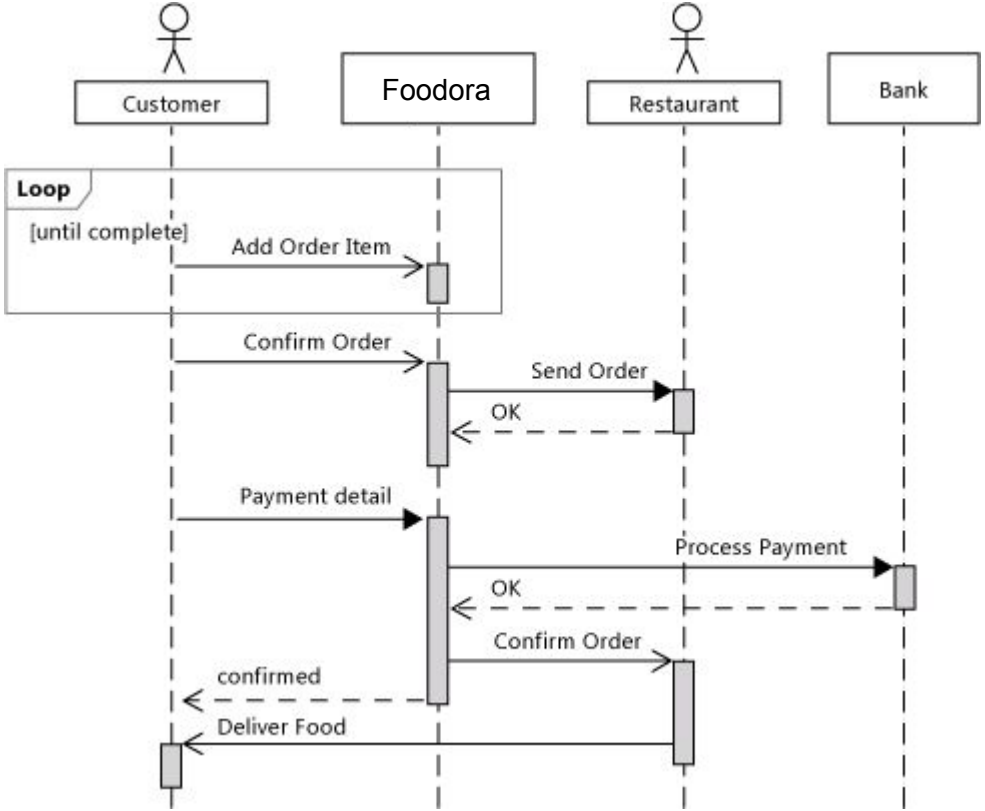
- Ikke alle sekvensdiagram har en *aktør*, mens andre kan ha flere.

- Kommunikasjon mellom objekter beskrives ved hjelp av *meldinger*. Disse illustreres som piler og opptrer i sekvensiell rekkefølge på livslinjen.



Sekvensdiagram

Eksempel



Type casting

- Kan eksplisitt konvertere en verdi fra én type til en annen type
- Typene man konverterer mellom må være relaterte i en eller annen form (enten arv eller implementasjon av grensesnitt)
- Man må gjøre **eksplisitt** casting dersom man konverterer fra en klasse eller et grensesnitt "høyere opp i hierarkitet":

```
Pet pet = new Dog("Fido");
Dog fido = (Dog) pet;
```

- Det trenger vi derimot *ikke* gjøre dersom vi går motsatt vei:

```
Dog fido = new Dog("Fido");
Pet pet = fido;
```