NTNU

Kunnskap for en bedre verden

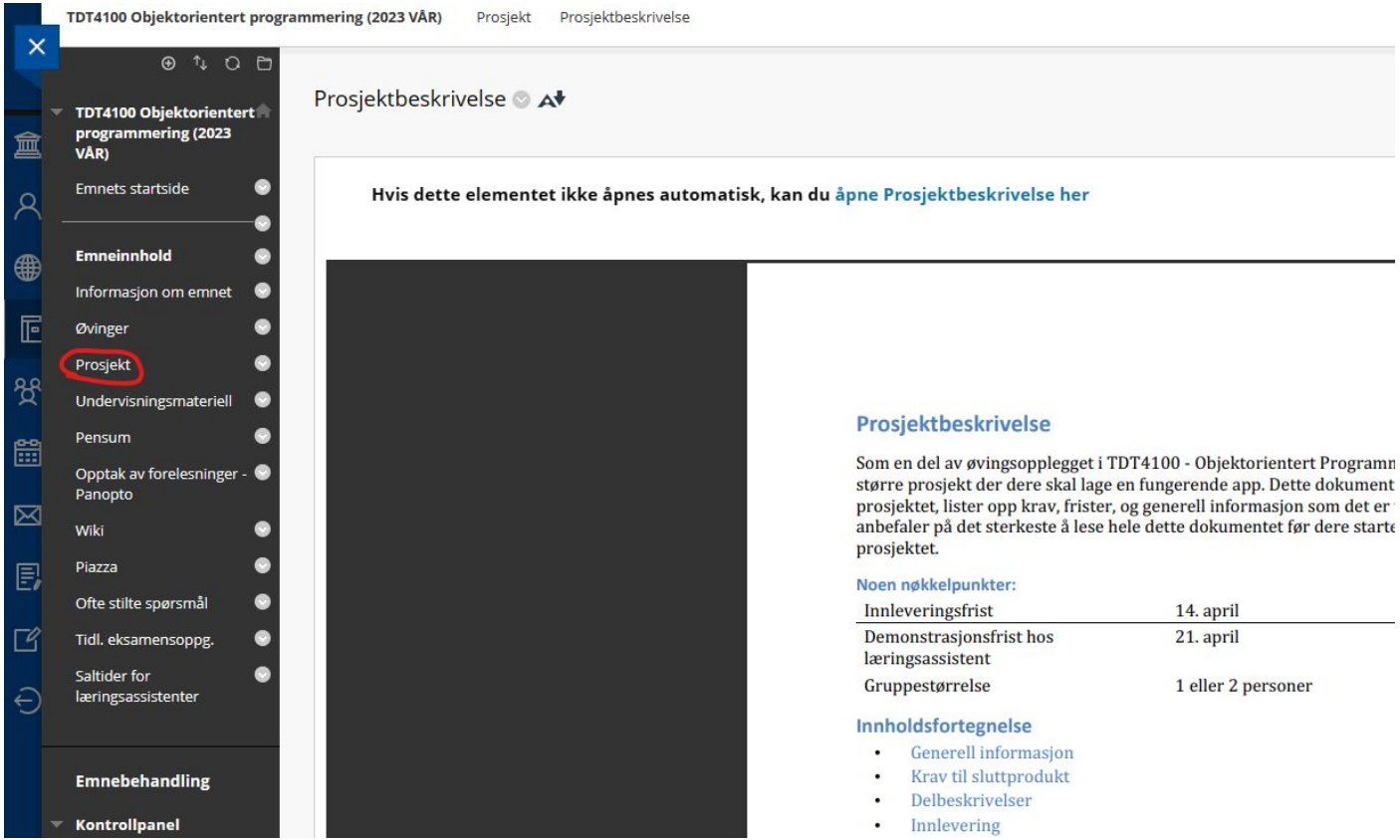
Øvingsforelesning 5 (del 2/2)

TDT4100 Objektorientert programmering

29.02.2022

Jostein Hjortland Tysse
Vitenskapelig assistent, TDT4100
jostein.h.tysse@ntnu.no

Prosjektbeskrivelse



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

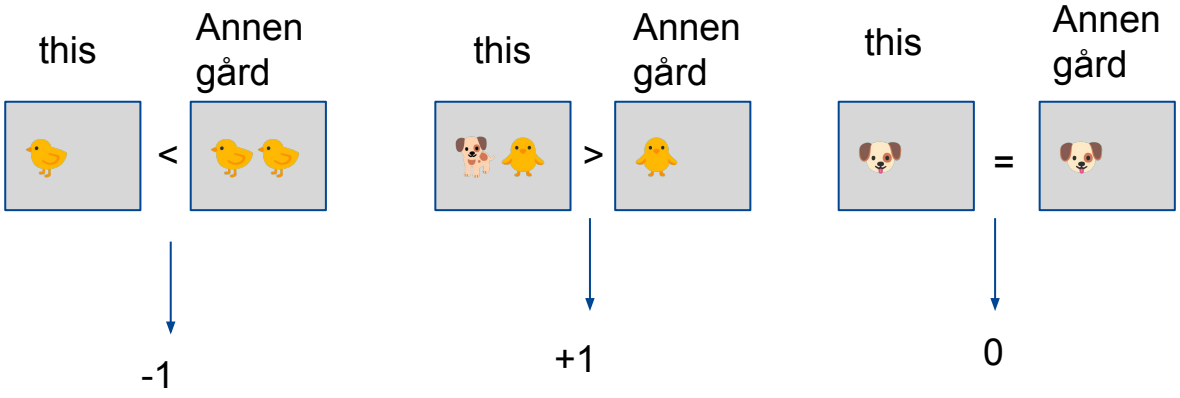
Praktisk oppgaveløsning

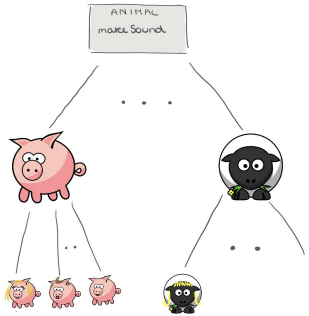
Vi fortsetter med **`Farm`** fra forrige uke

Oppgave 1 (repetisjon)

Vi ønsker å kunne sortere gårder (**Farm**-objekter) basert på hvor mange dyr som er på gården.

La Farm-klassen implementere Comparable-interfacet, og lag logikken i compareTo()-metoden.





Iterator og Iterable

Interfaces for iterasjon i java

Iterator og Iterable

- To interfaces som **kan** brukes for å gjøre livet litt enklere når man skal iterere over objekter som inneholder flere elementer (for eksempel en Collection, Liste, etc.)

- Itera**tor<T>**:
  - Et objekt som lar oss hente objekter av en spesifikk type **T** med **`next()`**-metoden, helt til det ikke er flere objekter å hente.
  - Kan sjekke om det er flere elementer igjen med **`hasNext()`**-metoden
  - Analogi: peker som bevegere seg sekvensielt gjennom en liste

- Itera**ble<T>**:
  - interface som kun har én metode:
    - **`iterator()`**, som returnerer et **Iterator**-objekt for objekter av typen **T**

Iterable

- Definerer at dette objektet er en type som kan **itereres over**, for eksempel med en **foreach**-løkke
- Vi vet **allerede** om en del typer 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:

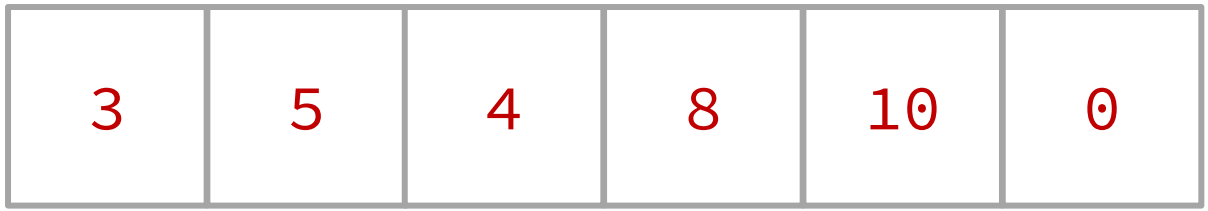
```
public class Kennel implements Iterable<Dog> {
	List<Dog> dogs;
	public Kennel(Dog[] dogList) {
		this.dogs = new ArrayList<>(Arrays.asList(dogList));
	}
	@Override
	public Iterator<Dog> iterator() {
		return /* Et iterator-objekt for typen Dog*/;
	}
}
```

- Vi kan nå iterere over **Dog**-objekter i objekter av typen **Kennel** slik:

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

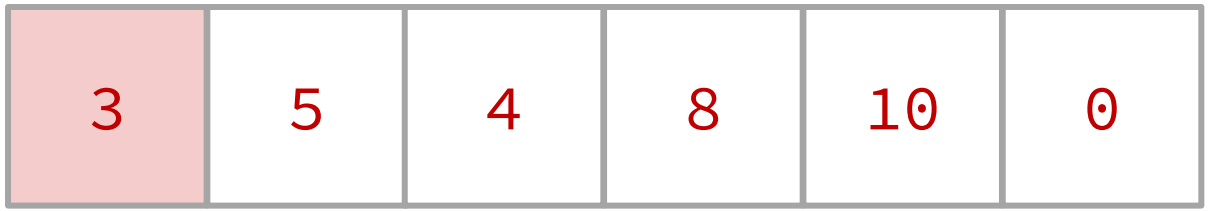
Iterator

Motivasjon



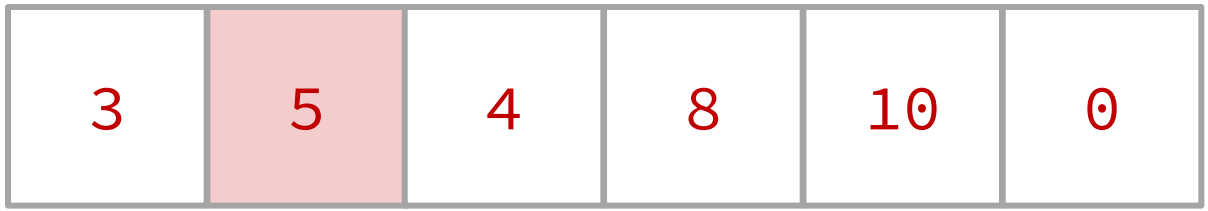
Iterator

Motivasjon



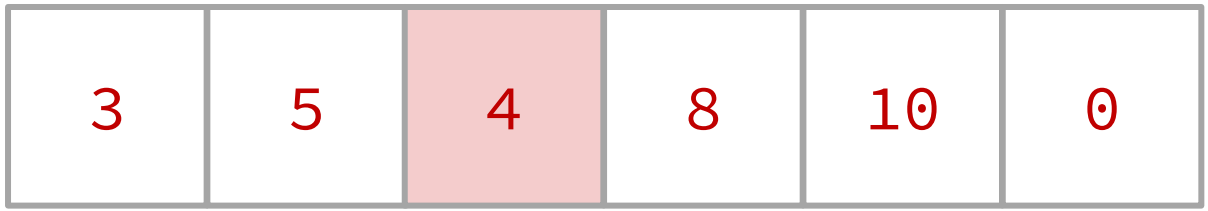
Iterator

Motivasjon



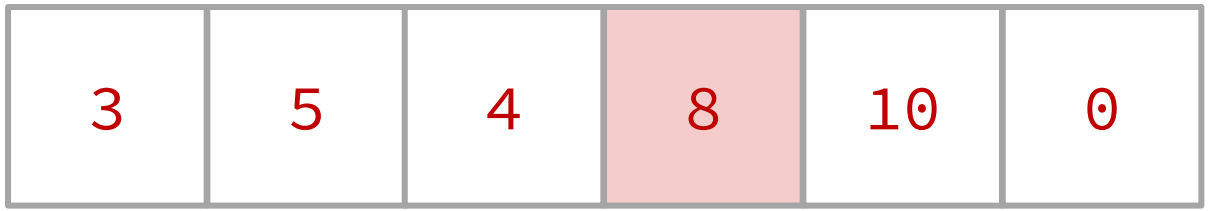
Iterator

Motivasjon



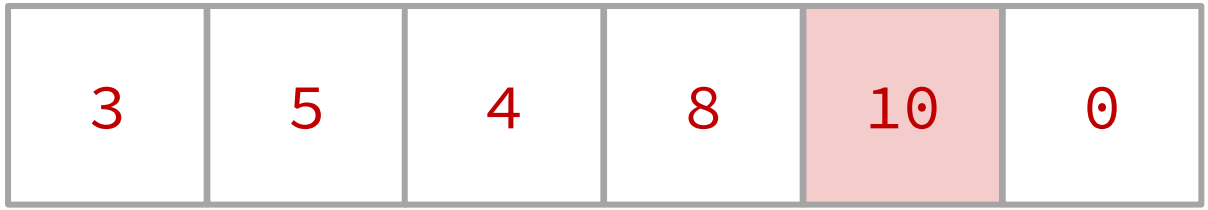
Iterator

Motivasjon



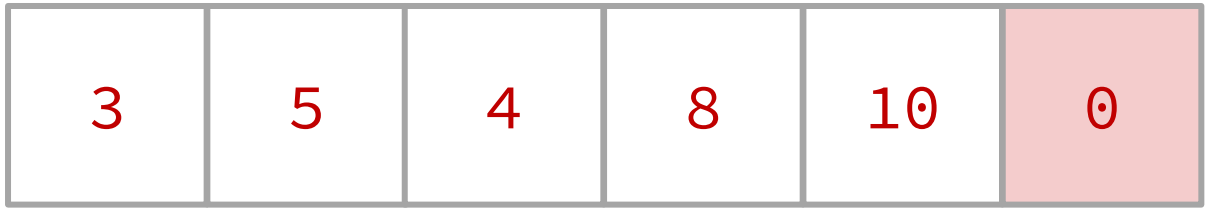
Iterator

Motivasjon



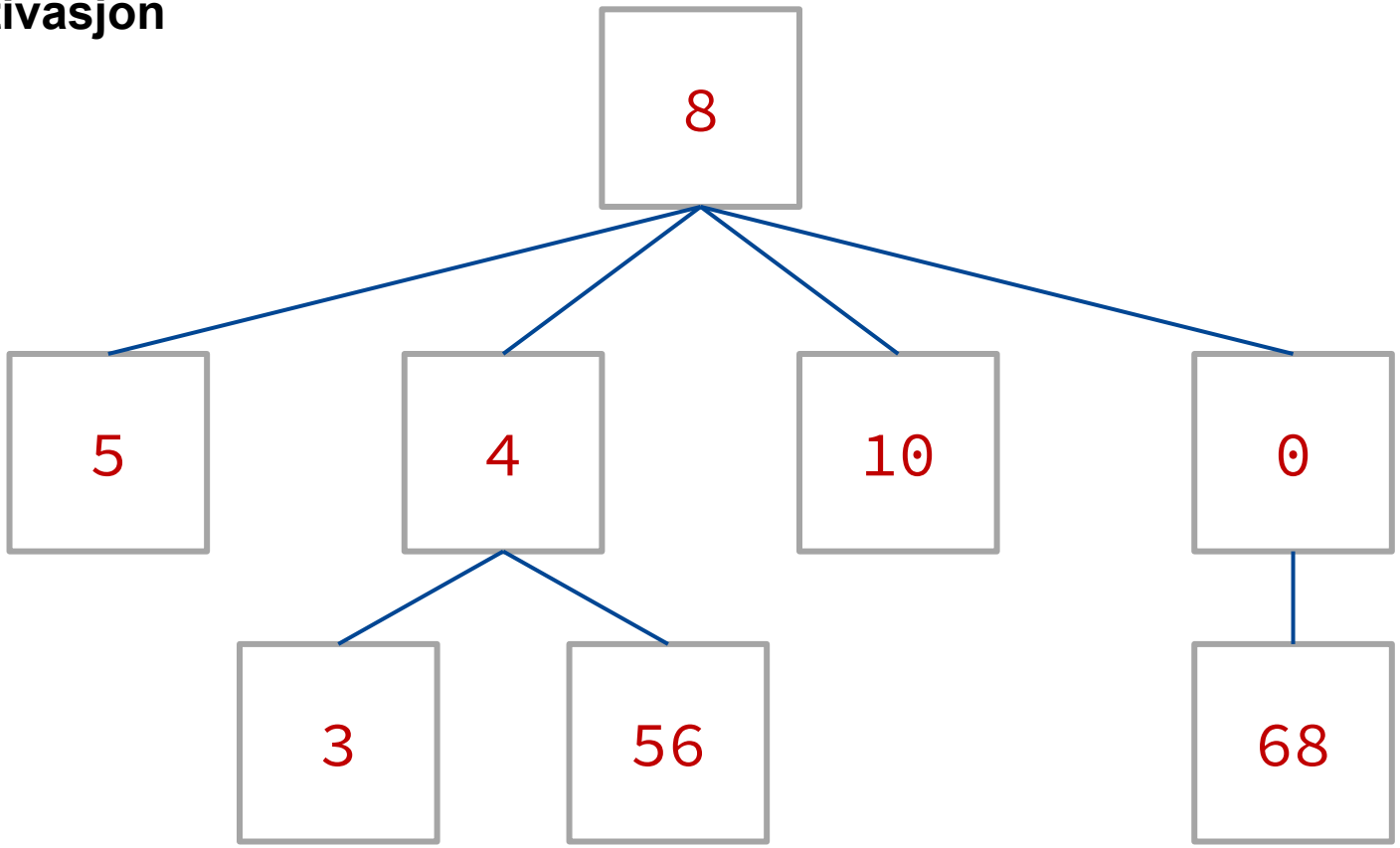
Iterator

Motivasjon



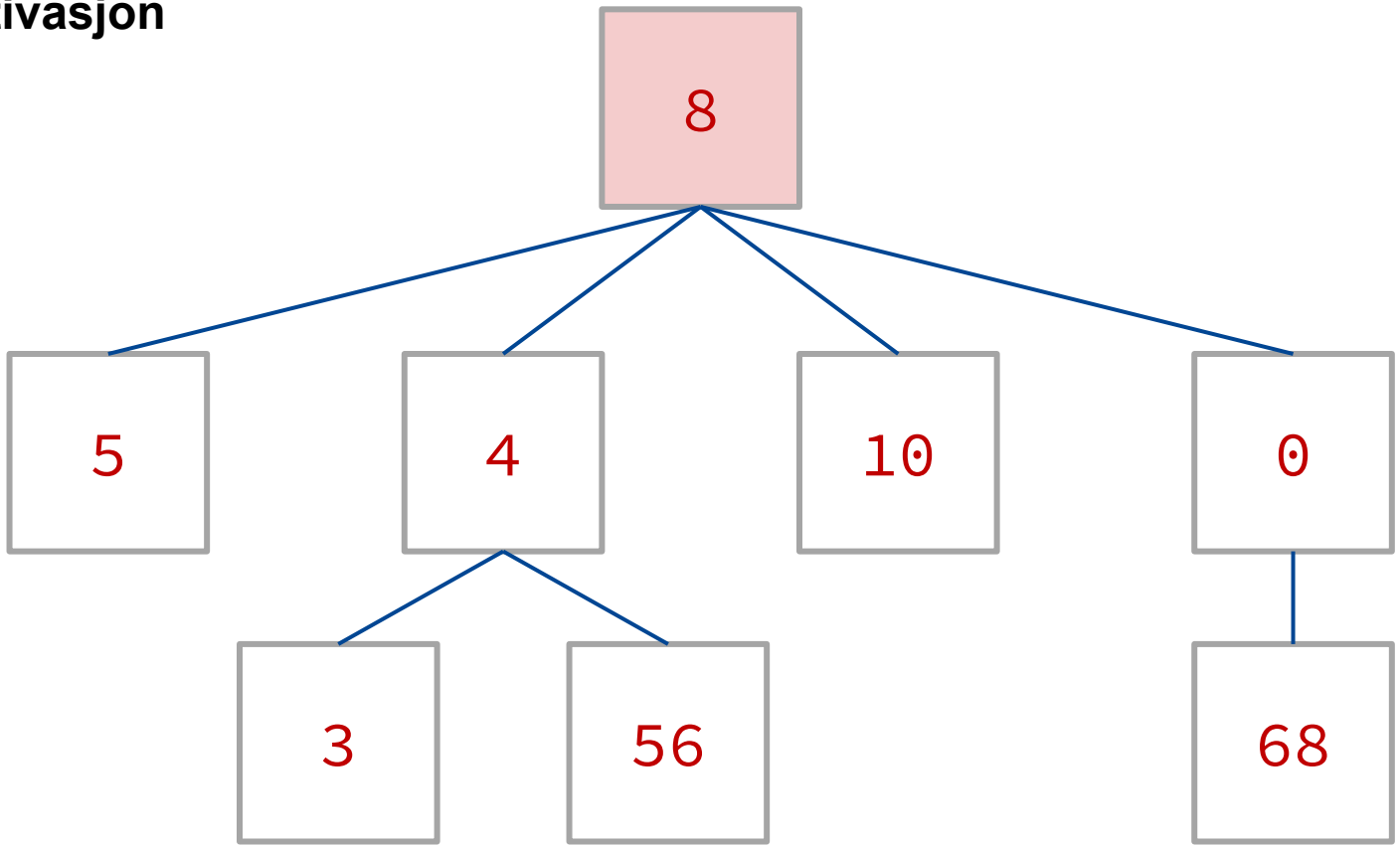
Iterator

Motivasjon



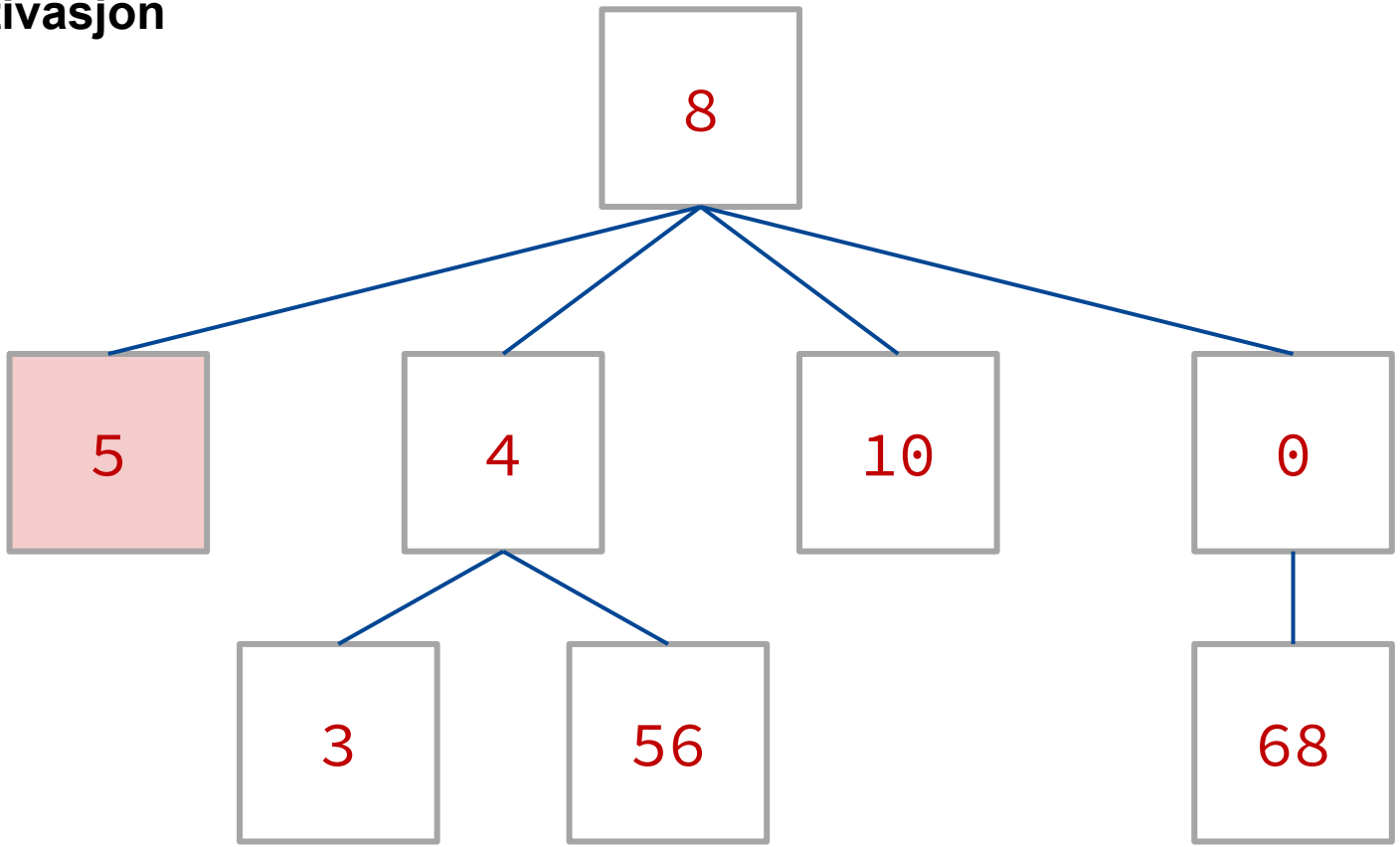
Iterator

Motivasjon



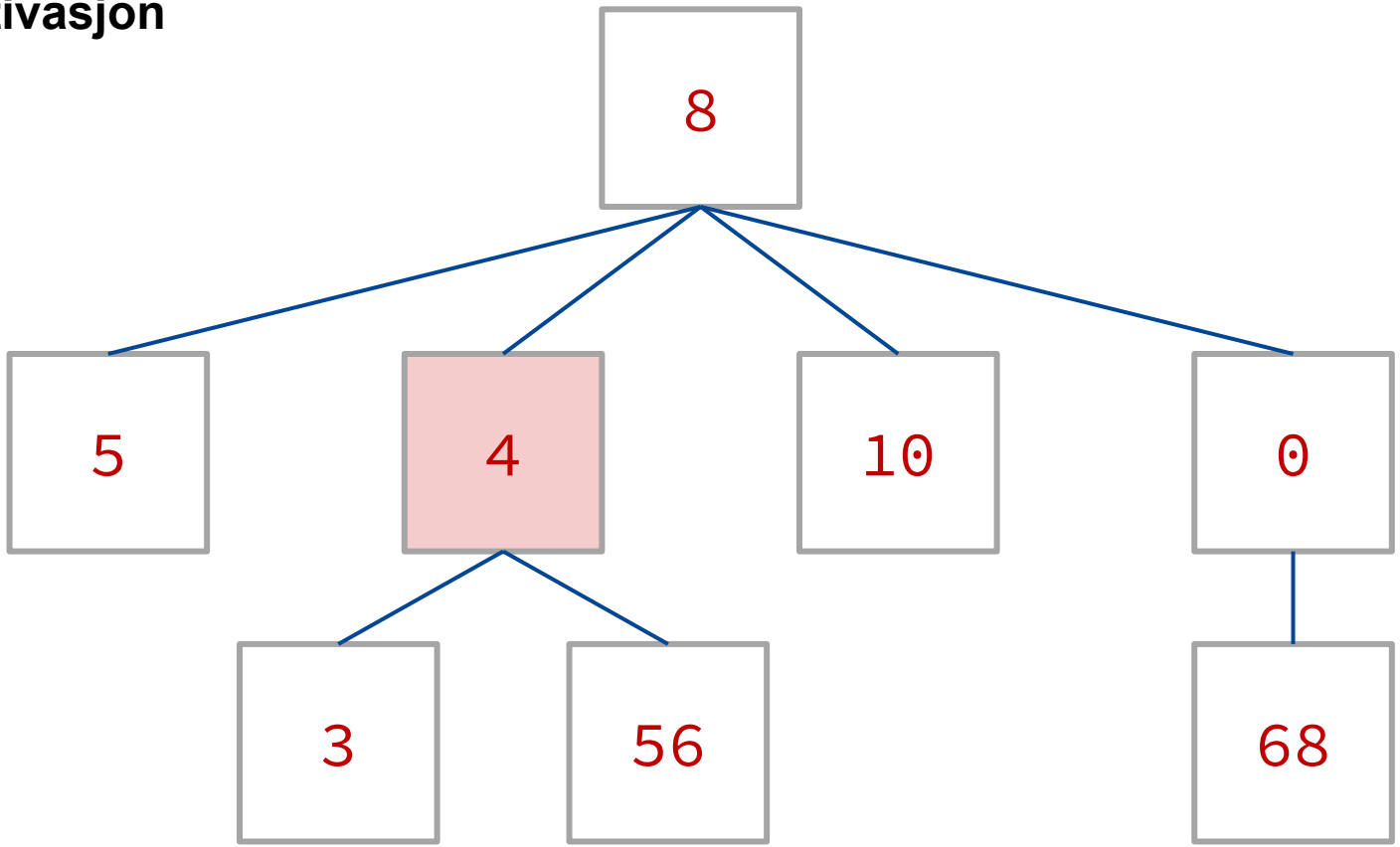
Iterator

Motivasjon



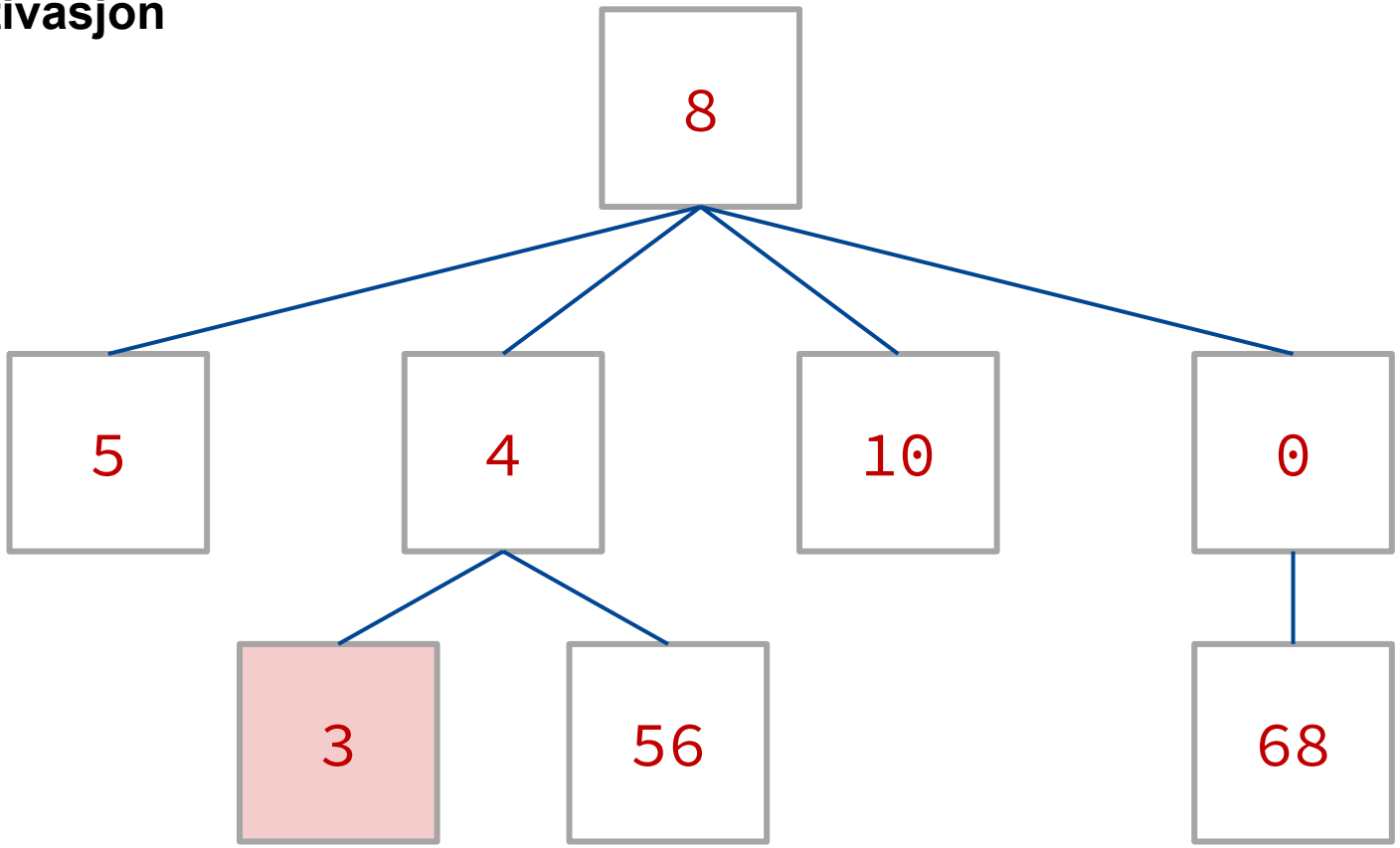
Iterator

Motivasjon



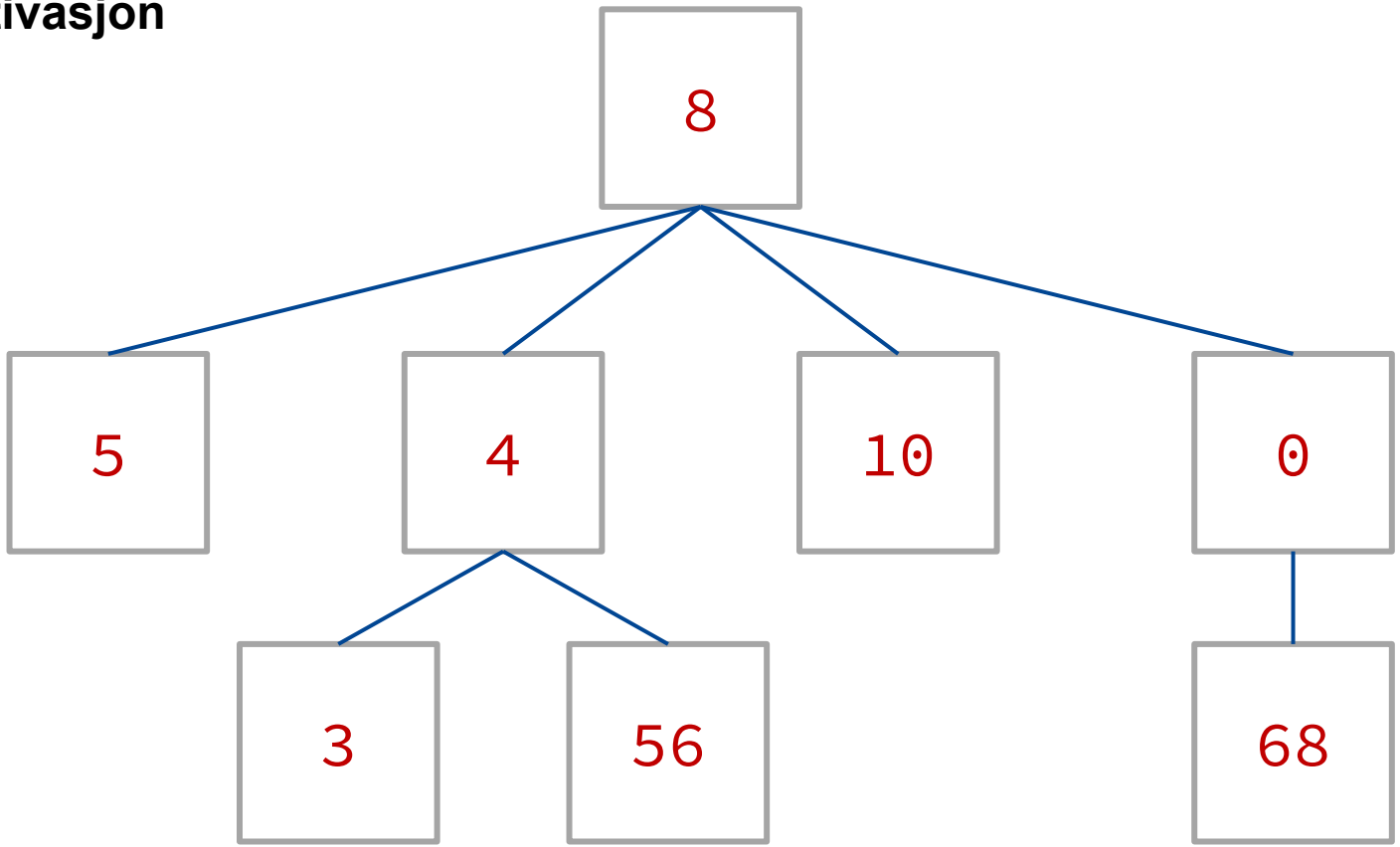
Iterator

Motivasjon



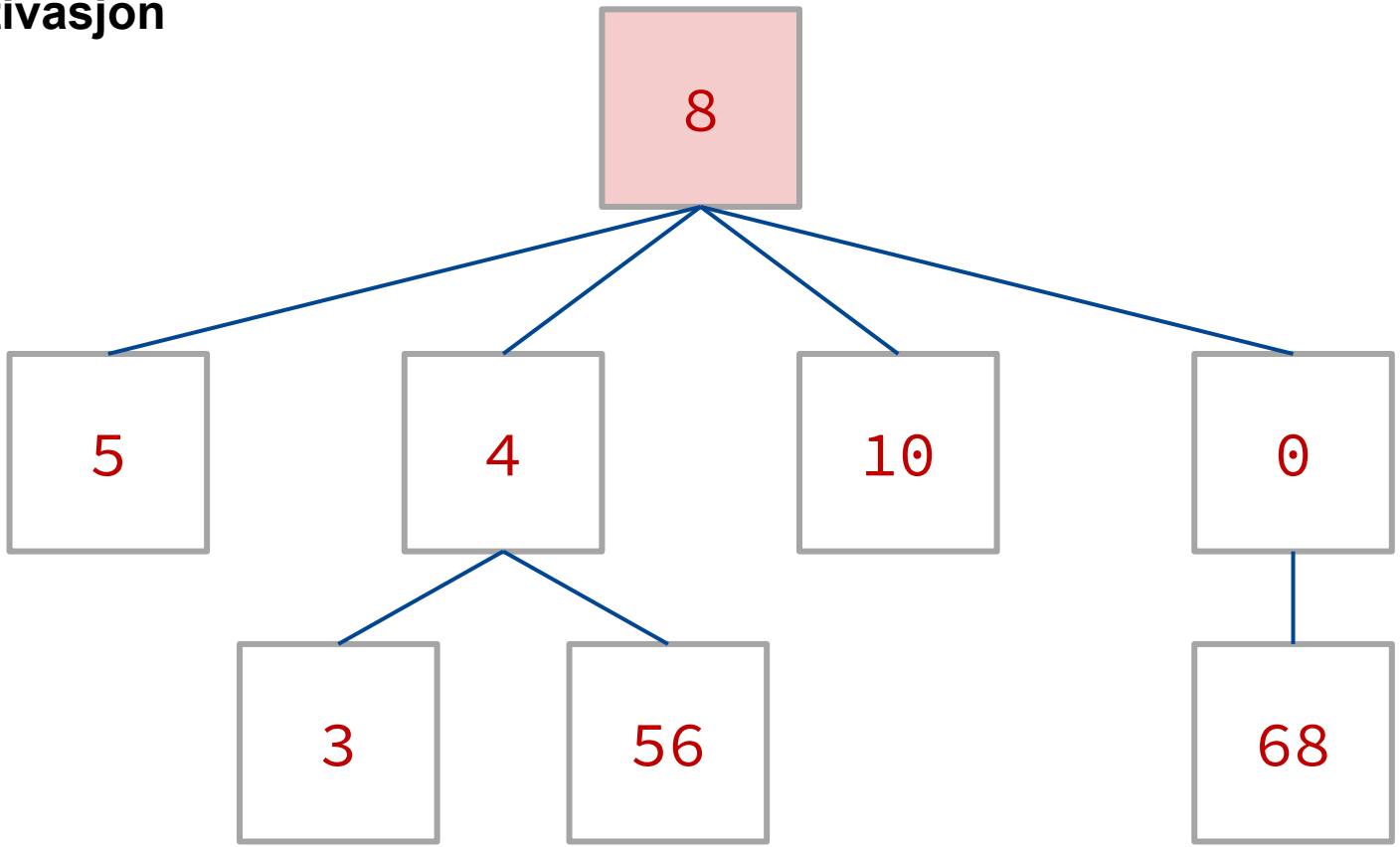
Iterator

Motivasjon



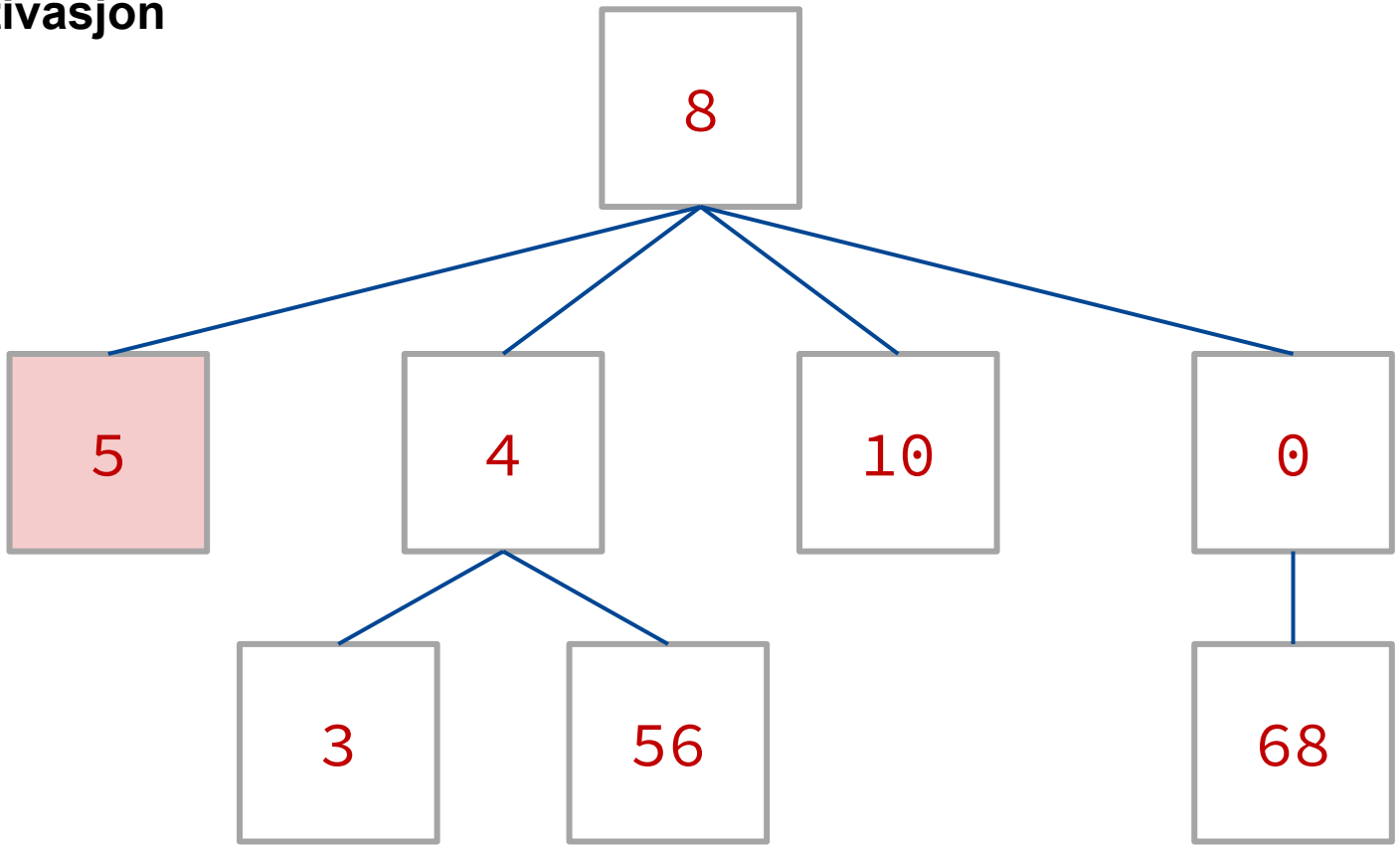
Iterator

Motivasjon



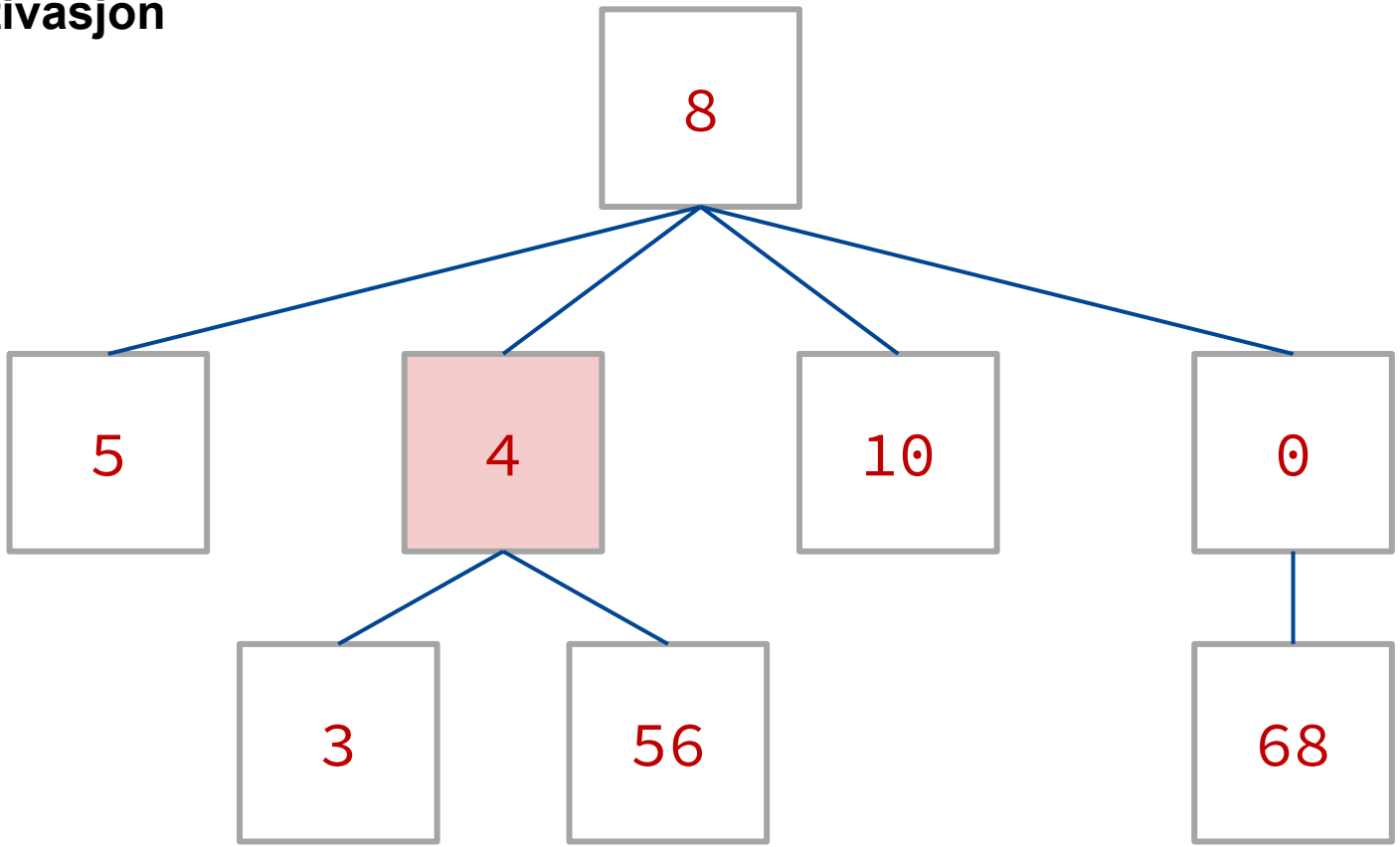
Iterator

Motivasjon



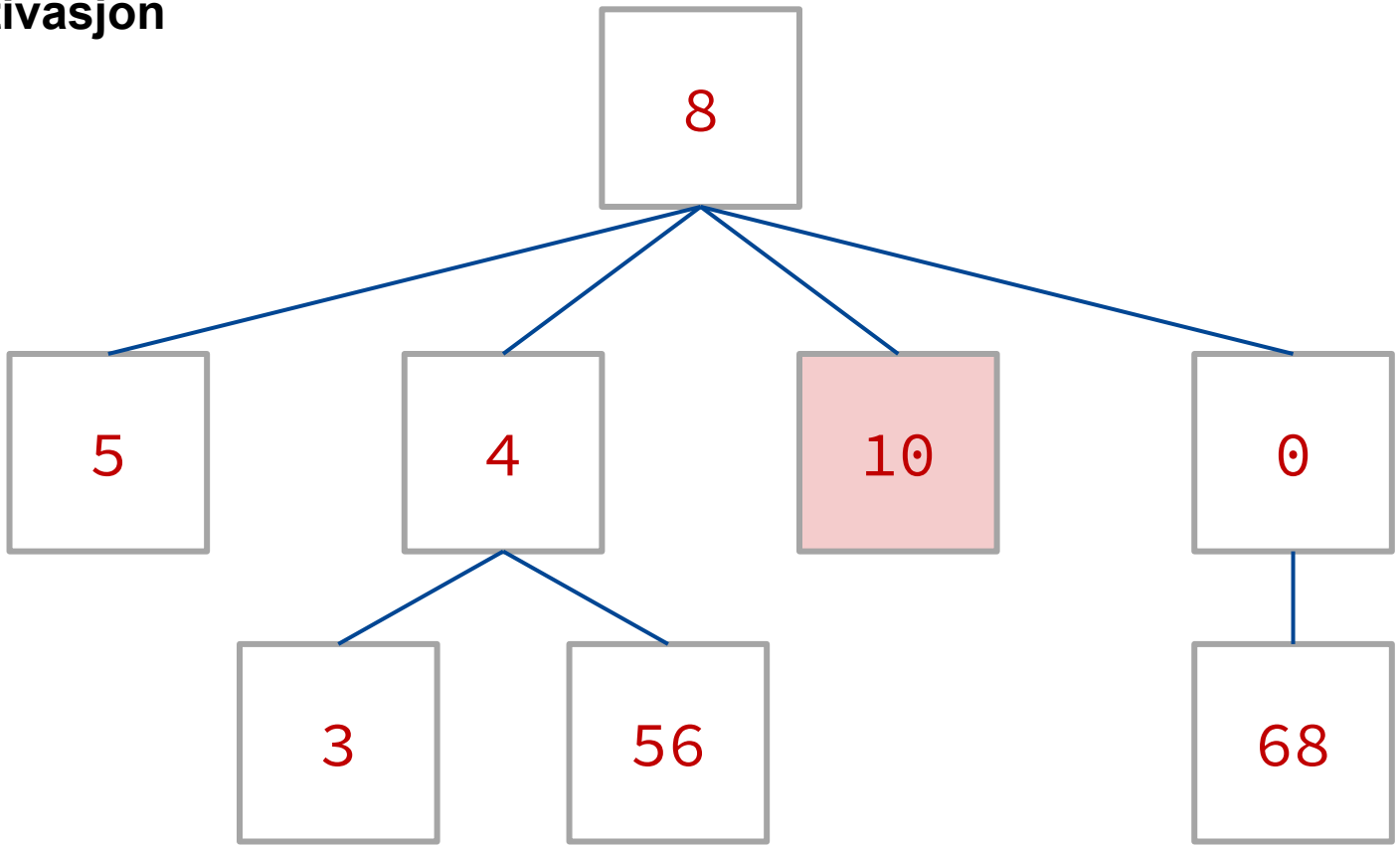
Iterator

Motivasjon



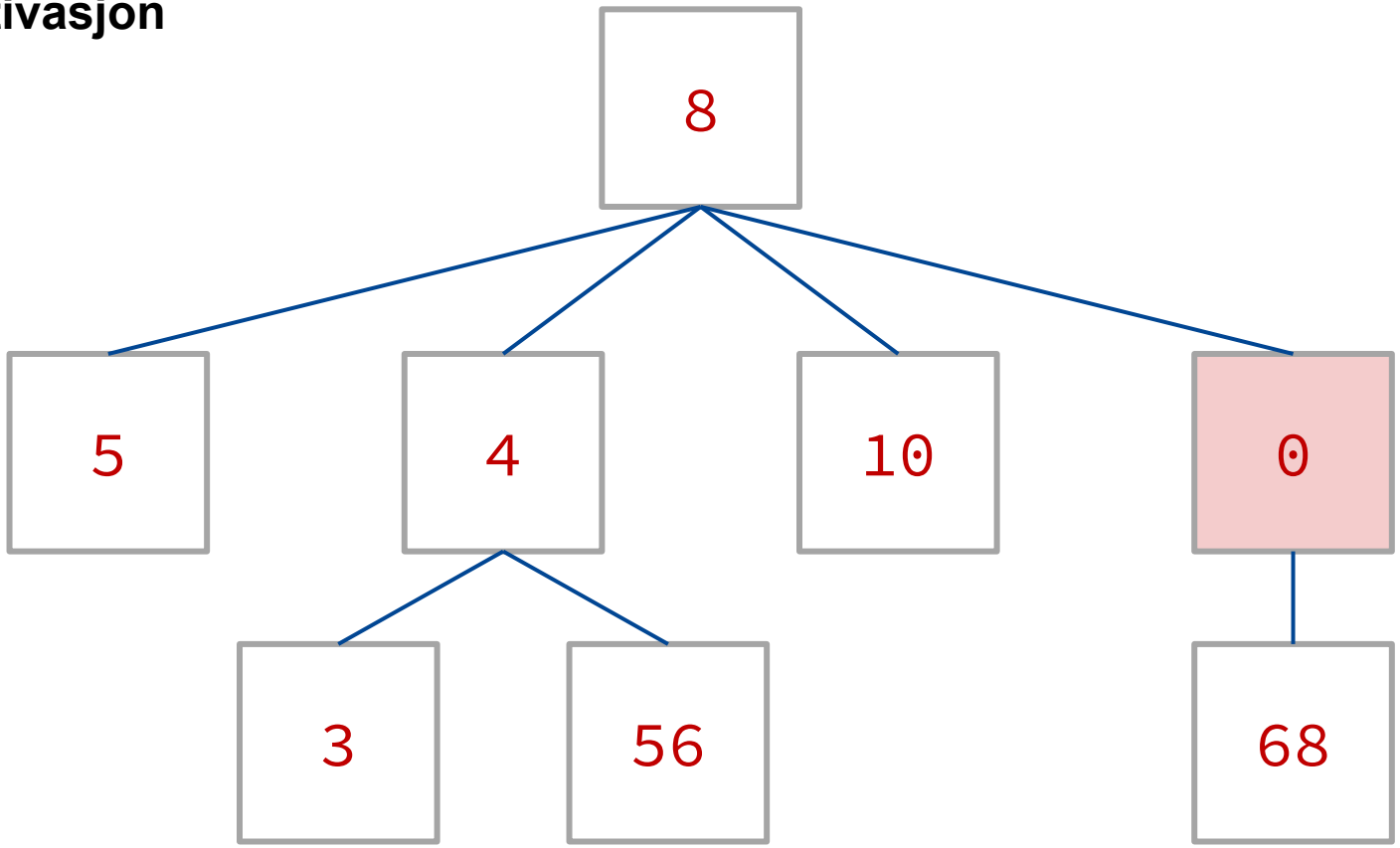
Iterator

Motivasjon



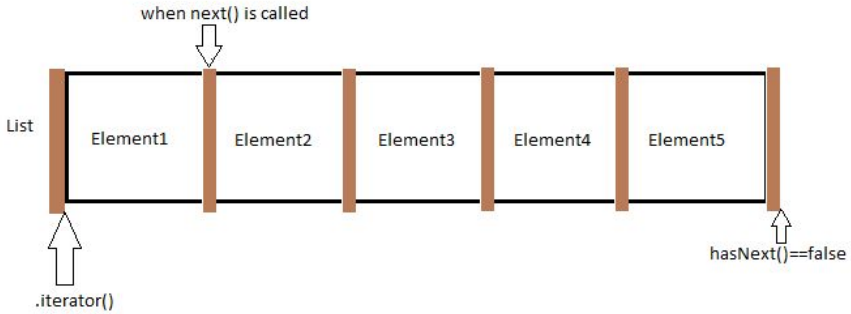
Iterator

Motivasjon



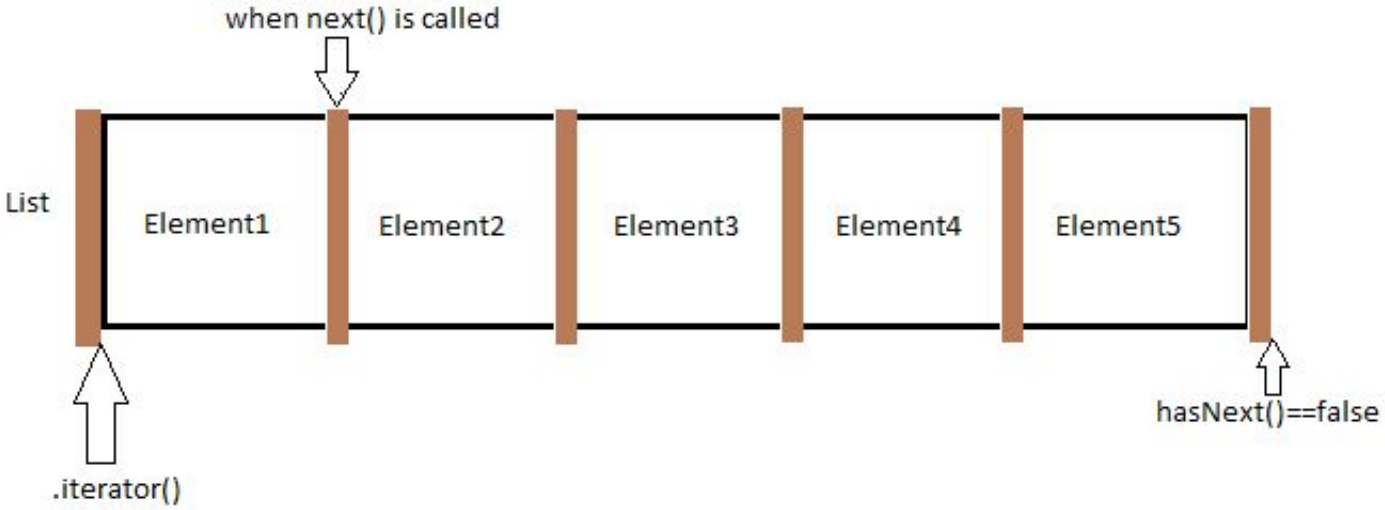
Iterator

- Et type objekt som **itererer** over en mengde objekter (utfører itereringen)
  - I motsetning til **`Iterable`**, som er det som **itereres over**
- Fungerer i stor grad likt som å iterere over en liste med en **foreach**-løkke, men vi kan definere rekkefølgen og regler for itereringen i større grad, samt at vi kan **fjerne elementer** fra lista **mens vi itererer**
- Implementasjon av dette interfacet krever at vi implementerer to metoder:
  - **`next()`**
  - **`hasNext()`**



Iterator-interfacet

Visualisering



Iterator-interfacet

Eksempel

```
public class ShoppingListIterator implements Iterator<Item> {
     private ShoppingList shoppingList;
     private int pos;
     public ShoppingListIterator(ShoppingList shoppingList) {
          this.shoppingList = shoppingList;
          this.pos = 0;
     }
     @Override
     public boolean hasNext() {
          return pos < shoppingList.getItemCount();
     }
     @Override
     public Item next() {
          Item item = shoppingList.getItem(pos);
          pos += 1;
          return item;
     }
```

Oppgave 2

Vi ønsker nå mulighet til å kun iterere over dyr **som er 2 år eller yngre i en Farm.**

Lag en klasse YoungAnimalsIterator som implementerer Iterator-interfacet. Det skal kun returneres Animal-objekter med en alder mindre enn eller lik 2 år

main-metoden i filen Oppgave2.txt kan kjøres for å teste implementasjonen din.

Iteratorer og `Collection`-rammeverket

- Alle klasser som implementerer **`Collection`**-grensesnittet (f.eks. **`List`**) implementerer **`Iterable`**-grensesnittet og har allerede en ferdig definert **`iterator()`**-metode.
  - Dette kan vi se ut ifra at dette er gyldig syntaks:

```
List<Dog> dogList = new ArrayList<>();
for(Dog dog : dogList) {
    dog.bark();
}
```

  - Husk at for å iterere med **foreach** over et objekt så er dette objektet nødt til å implementere **`Iterable`**

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

NTNU

Funksjonelle grensesnitt

+ Streams og lambda-uttrykk

Obs! Viktig å nevne om det vi skal lære nå

- **Ikke få panikk** selv om du ikke henger med, for mange vil grensesnitt (og relatert pensum) være noe av det det vanskeligste i pensum.
  - Forstår du bare litt her nå i dag så er du fortsatt kommet langt.

- Dersom du synes mye av det mer grunnleggende er vanskelig (for eksempel generell bruk av grensesnitt) så kan det anbefales å fokusere på det før du går videre med dagens tema.

Funksjonell programmering

Tema for dagen

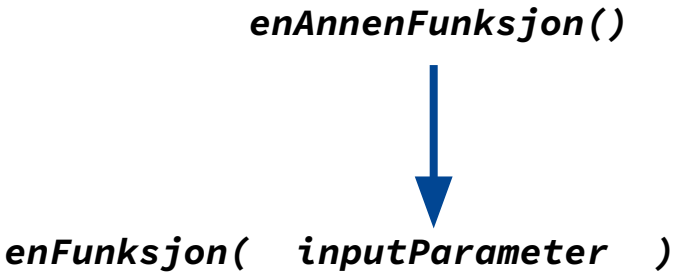
- Vi beveger oss nå inn på området **funksjonell programmering,** som har kommet inn i pensum i emnet de siste årene da det også er blitt mer sentralt i Java.
- Noen av oppgavene på **øving 5** introduserer disse temaene
- Funksjonell programmering kom inn i Java 8 og introduserer en del **nye konsepter** og **syntaks**.
- Dette er et eget **programmeringsparadigme** i seg selv, på samme måte som objektorientert programmering. I moderne programmering brukes som regel disse i kombinasjon med hverandre

```
employees.stream()
         .filter(emp -> emp.age < age)
         .forEach(emp -> System.out.println(emp));
```

Funksjonelle grensesnitt

Java benytter **funksjonelle grensesnitt** for å la oss **instansiere funksjoner** som **objekter** i Java.

- Hensikten med dette er, blant annet, å kunne ta inn **funksjoner** som input-argument i andre funksjoner igjen!



Funksjonelle grensesnitt

Funksjonelle grensesnitt er egentlig veldig like grensesnittene vi har sett til nå, men for at et grensesnitt skal være definert som **funksjonelt** så stilles det imidlertid noen krav*:

- Funksjonelle grensesnitt har kun definert én (abstrakt) metode
  - *Dette er Java sitt eneste formelle krav*
- Objektet som implementerer grensesnittet har nettopp dette som sin primære funksjon, og ikke primært noe annet
- Metoden som implementeres er ikke avhengig av å bruke intern tilstand

Vi kan derfor ikke avgjøre hvorvidt et grensesnitt egentlig er funksjonelt basert kun på å se på selve grensesnittet, vi må også se hvordan det benyttes av andre klasser som implementerer det.

*Dette ble stilt som spørsmål på bla. eksamen V2017

Funksjonelle grensesnitt:

`Comparator` vs. `Comparable`

- `Comparator` er et funksjonelt grensesnitt, men `Comparable` regnes ikke som det, selv om det også det er et grensesnitt med bare én metode.

- Grensesnittet `Comparable` er ment å bli implementert av data-klasser som f.eks. `Person`, `Card` og lar objektene sammenligne seg selv med et annet av samme type.
  - Objektet er da **primært** et data-objekt, og implementerer grensesnittet som en **sekundær** funksjon.
  - Det ville ikke gitt mening å implementere `Comparable` i en klasse som ikke har noen annen funksjon – da er det jo ingenting å sammenligne!

- `Comparator` kan imidlertid implementeres av en klasse som har det som sin **primære** funksjon (f.eks. en `CardComparator`)

Eksempel på klasse som implementerer `Comparable`

- Klassen **`Medication`** implementerer grensesnittet **`Comparable`**, men har samtidig mange andre funksjoner
- **`Comparable`** kan dermed **teknisk sett** kalles et funksjonelt grensesnitt, men i praksis er det ikke mulig å implementere det på en hensiktsmessig måte uten å ha noe å sammenligne med.

```java
public class Medication implements Comparable<Medication> {

    private String name;
    private double price = Double.NaN;

    public Medication(String name) {
        this.name = name;
    }

    public Medication(String name, double price) {
        this.name = name;
        if(price < 0) throw new IllegalArgumentException("Invalid price");
        this.price = price;
    }

    public String getName() {
        return name;
    }

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

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    // Comparable-grensesnittet:
    @Override
    public int compareTo(Medication medication) {
        return Double.compare(this.getPrice(), medication.getPrice());
    }

    @Override
    public String toString() {
        return this.getName();
    }
```

`Comparator` som funksjonelt grensesnitt

- Klasser som implementerer **`Comparator`**-grensesnittet har i motsetning til **`Comparable`** ingen annen funksjon enn å sammenligne to objekter av samme type.

- Derfor oppfyller det kravet om at klasser som implementerer grensesnittet har dette som sin primære funksjon, og **`Comparator`** er derfor et **funksjonelt grensesnitt**.

Eneste metode i **`PersonAgeComparator`**

```
public class PersonAgeComparator implements Comparator<Person>{
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getAge() - o2.getAge();
    }
}
```

Funksjonelle grensesnitt
Motivasjon

- Hvorfor så mye fokus på **grensesnitt** med kun **én metode**?
- Funksjonelle grensesnitt lar oss benytte oss av noe som kalles **lambda-uttrykk**
    - Lamda-uttrykk **representerer funksjoner**, og er en annen måte å skrive dette på.
    - Slipper å definere typer når vi bruker lambdaer, i motsetning til ellers i Java
    - Vi bruker disse for å **implementere** og samtidig **instansiere** funksjonelle grensesnitt
    - Ny syntaks, men i all hovedsak **samme logikk** som man er vant med fra før:

```
(input -> input > 5)
```

Funksjonelle grensesnitt
Motivasjon

- Hvorfor så mye fokus på **grensesnitt** med kun **én metode**?
- Funksjonelle grensesnitt lar oss benytte oss av noe som kalles **lambda-uttrykk**
    - Lamda-uttrykk **representerer funksjoner**, og er en annen måte å skrive dette på.
    - Slipper å definere typer når vi bruker lambdaer, i motsetning til ellers i Java
    - Vi bruker disse for å **implementere** og samtidig **instansiere** funksjonelle grensesnitt
    - Ny syntaks, men i all hovedsak **samme logikk** som man er vant med fra før:

```
(input -> input > 5)
```

Lambda-uttrykk
Eksempel

```
// Lambda-uttrykk:

(input -> doSomethingWith(input))
```

```
// Vanlig metode:

public Object funksjonsNavn(Object input) {
    // Code
    return doSomethingWith(input);
}
```

Lambda-uttrykk
Eksempel

```
// Lambda-uttrykk:

(input) -> doSomethingWith(input)
```

```
// Vanlig metode:

public Object funksjonsNavn(Object input) {
    // Code
    return doSomethingWith(input);
}
```

Lambda-uttrykk

Eksempel

```
// Lambda-uttrykk:
(input -> {
    // Code
    doSomethingWith(input)
})
```

```
// Vanlig metode:

public Object funksjonsNavn(Object input) {
    // Code
    return doSomethingWith(input);
}
```

Lambda-uttrykk

Eksempel

```
// Lambda-uttrykk:
(input) -> {
    // Code
    doSomethingWith(input)
})
```

```
// Vanlig metode:

public Object funksjonsNavn(Object input) {
    // Code
    return doSomethingWith(input);
}
```

Lambda-uttrykk
Motivasjon

- Hva er så spesielt med **lambda-uttrykk**, hvorfor trenger vi disse?
- Lambda-uttrykk lar oss benytte oss av **Streams**, i tillegg til mange andre operasjoner som kan gjøre livet enklere i Java.
- Lambda-uttrykk er i praksis funksjoner som kun tar inn argumenter og gjør noe med disse.
    - Det er altså ingen intern tilstand som benyttes
- Mindre **boilerplate**-kode
    - Les: mindre kode som må skrives for å oppnå mer.

Lambda-uttrykk:

`.forEach()`

- Vi er vant med å skrive foreach-løkker slik:

```
for(String ord : ordListe) {
    System.out.println(ord);
}
```

- Ved å bruke den innebygde **`.forEach()`**-metoden i **`Collection`**-rammeverket kan vi skrive det på denne måten i stedet:

```
ordListe.forEach(ord -> {
    System.out.println(ord)
});
```

Funksjonelle grensesnitt:

`Predicate`-grensesnittet

- Et predikat sier om et objekt oppfyller et visst kriterium

- **`Predicate`**-grensesnittet har én metode som heter **`test()`** og som tar inn et objekt av hvilken som helst type som argument, og returnerer en **`boolean`**. Altså ganske likt som å skrive et **boolsk uttrykk** (det vi bruker i **`if`**-setninger)

- Grensesnittet er ment å brukes til å **teste** hvorvidt en **påstand** om et objekt er sant eller ikke. Dette får vi spesielt nytte for dersom vi for eksempel ønsker å **filtrere** en liste basert på en betingelse

NTNU

`Predicate`-grensesnittet:

Hvorfor *funksjonelt*?

- Strengt tatt så kan man ikke sette funksjoner som objekt i Java.
- Vi kan derimot instansiere klasser med *kun én* metode:

```
public class TestAgePredicate implements Predicate<Integer>
{
    public boolean test(Integer age) {
        return age.intValue() > 5;
    }
}
```

- Med Lambda-uttrykk så kan dette uttrykkes slik i stedet:

```
Predicate<Integer> testAgePredicate = (age -> age > 5);
```

NTNU

`Predicate`-grensesnittet:
Hvorfor *funksjonelt*?

- Merk at en klasse som implementerer **`Predicate`** kun trenger å implementere <u>én metode</u> og har nettopp det å implementere dette grensesnittet som sin <u>primære funksjon</u>

```
public class TestAgePredicate implements Predicate<Integer>
{
    public boolean test(Integer age) {
        return age.intValue() > 5;
    }
}
```

`Predicate`-grensesnittet:

Kontekst

- Fra Wikipedia:

> In traditional grammar, a **predicate** is one of the two main parts of a sentence (the other being the subject, which the predicate modifies). For the simple sentence "John [*is yellow*]", *John* acts as the subject, and *is yellow* acts as the predicate, a subsequent description of the subject headed with a verb.

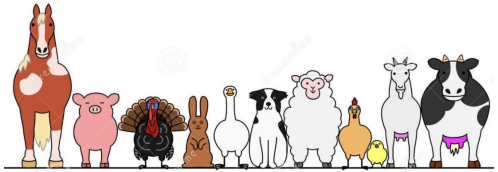
- Innenfor grammatikken kommer et **predikat** i en setning med en **påstand** om subjektet:
  - "Jens ***[gjorde øvingen sin]***"
- Hvis vi ser på `Jens` som subjektet nedenfor, så ser vi at `Jens` ***`hasDoneExercise()`*** er predikatet i setningen:

```
(Jens -> Jens.hasDoneExercise())
```

Oppgave 3

Hvis vi i **Farm** ønsker å filtrere på noe annet enn alderen til dyrene, må vi per nå skrive en helt ny iterator. Dette kan være tungvint hvis vi vil gjøre mange forskjellige filtreringer.

Lag en klasse *FilterAnimalsIterator* som fungerer på samme måte som *YoungAnimalsIterator*, med unntak av at den bruker en implementasjon av grensesnittet *Predicate* til filtrering. Lag også en klasse *YoungAnimalPredicate* som implementerer *Predicate* og sjekker om dyret er under 2 år.
Ta inn Predicate i konstruktøren til iteratoren.



main-metoden i filen Oppgave3.txt kan kjøres for å teste implementasjonen din. Ved riktig implementasjon skal det samme som i oppgave 2 skrives ut

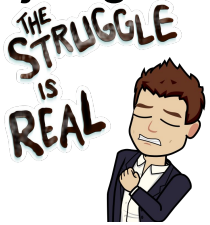
Oppgave 4

Vi kan nå utnytte **FilterAnimalsIterator** til å filtrere på andre ting enn kun alder.

Lag en klasse *DogPredicate* som implementerer *Predicate* og tester om dyret er av typen hund.

Oppgave 5

Det er litt tungvint å opprette en ny klasse for hvert forskjellig predikat vi ønsker å bruke.



Bruk Lambda-uttrykk for å skrive predikatet rett i konstruktøren til FilterAnimalsIterator i stedet for å bruke klassene fra de 2 forrige oppgavene.

Introduksjon til Streams i Java

- Dette er ikke det samme som **I/O-streams** i Java (filhåndtering)

- Streams gjør det veldig enkelt for oss å utføre operasjoner på lister på en kort og elegant måte (**one-liners**).

- Streams kan sammenlignes med å legge elementene i en liste/collection på et samlebånd og så gjøre én og én operasjon på hvert enkelt element

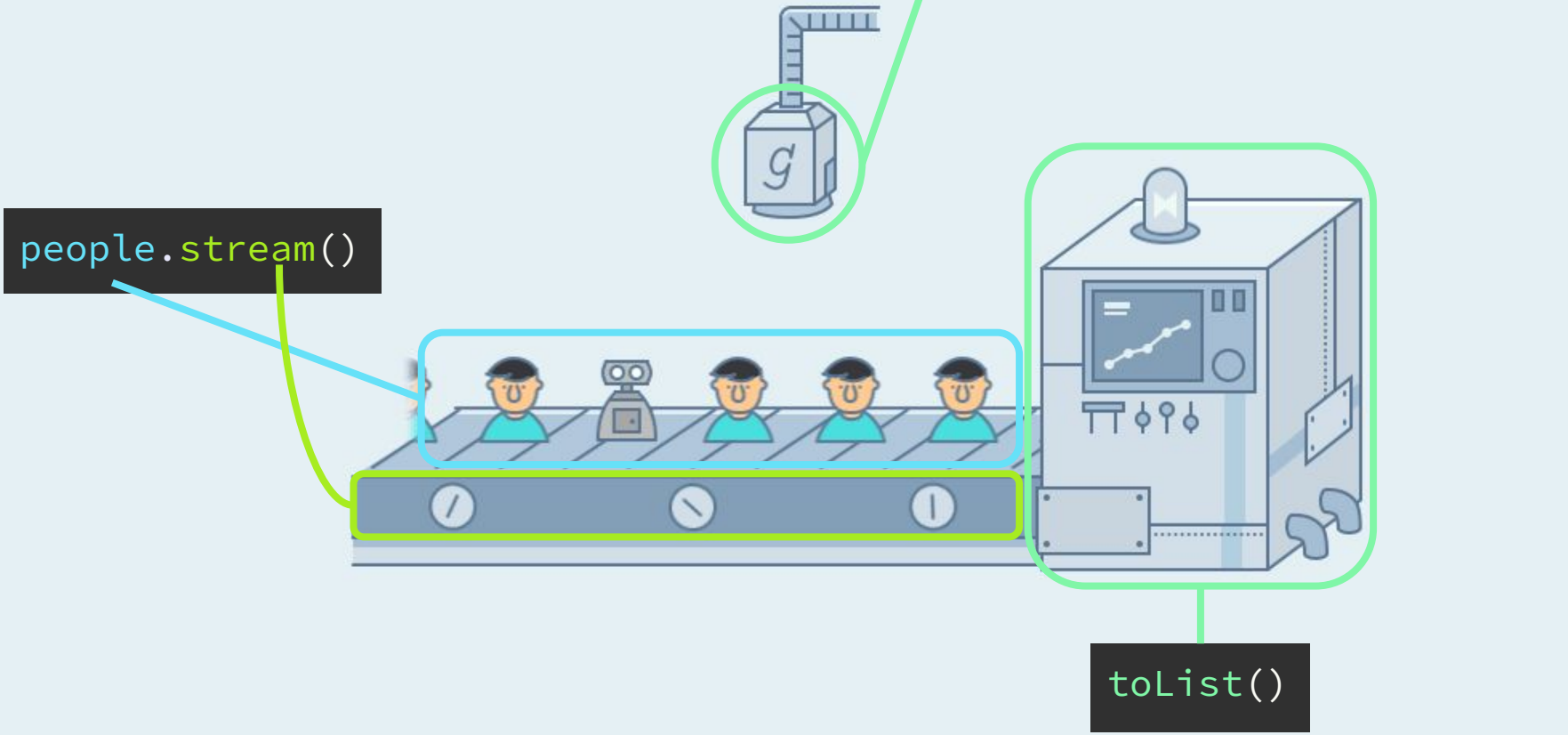
Streams:

Eksempel

- Koden nedenfor viser et eksempel på bruk av streams til å filtrere en liste
  - Dette tar inn en liste med objekter som implementerer **`Employee`**-grensesnittet og returnerer en ny liste med kun de som også er av typen **`Boss`**

```
List<Employee> employees = new ArrayList<>();
List<Employee> filteredEmployeeList =
          employees.stream()
                   .filter(e -> e instanceof Boss)
                   .toList();
```

```
people.stream()
        .filter(p -> p instanceof Human)
        .toList();
```



Streams
Nyttige metoder

- Vi går gjennom noen av de mest vanlige metodene man kan bruke i Streams:
  - **`anyMatch()`**
  - **`map()`**
  - **`distinct()`**
  - **`toList()` / `collect()`**
- Husk at alle disse kan brukes i kombinasjon med hverandre og lar oss gjøre forskjellige oppgaver eller tester i sekvens
- I tillegg til disse kan vi også bruke (bla.) metodene **`filter()`** og **`forEach()`** med streams

Nyttige metoder i Stream:

anyMatch()

- Sjekker om det finnes minst ett element i listen som oppfyller kriteriet
  - Tar inn et **Predicate**
- Måtte vanligvis gjort det på en (lignende) måte som til høyre.
- Kan gjøres på én linje med streams og **anyMatch()**:

```
boolean hasOldPerson = false;
for (Person p : persons) {
    if (p.getAge() >= 80) {
        hasOldPerson = true;
        break;
    }
}
System.out.println(hasOldPerson
);
```

```
System.out.println(persons.stream().anyMatch(p -> p.getAge() >= 80));
```

`anyMatch()`-funksjonen:

Visualisering

```
.anyMatch(animal -> animal instanceof Cat)
```

Strøm med
Animal-objekter

`.anyMatch()`

`true`

En boolean returneres hvis verdi er avhengig av om det finnes et objekt i strømmen som oppfyller predikatet

NTNU

Nyttige metoder i Stream:

`map()`

- Gjør en operasjon på hvert enkelt objekt, for eksempel å konvertere fra ett format til et annet.
- Tar inn en funksjon som tar inn et parameter og returnerer noe (**`Function`**-grensesnittet)

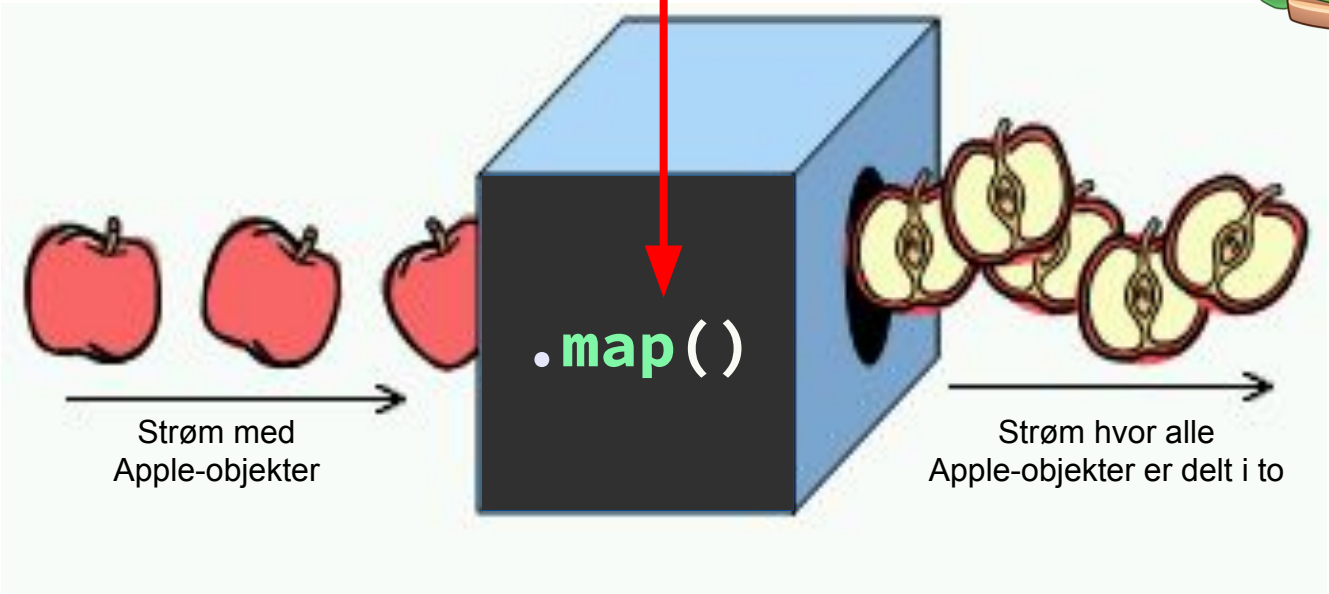
```
List<Person> persons = new ArrayList<Person>();
List<Integer> ages = new ArrayList<Integer>();

for (Person p : persons) {
    ages.add(p.getAge());
}

System.out.println(ages);
```

```
System.out.println(persons.stream().map(p -> p.getAge()).toList());
```

`map()`-funksjonen:

Visualisering

```
.map(apple -> apple.slice())
```



Nyttige metoder i Stream:

`distinct()`

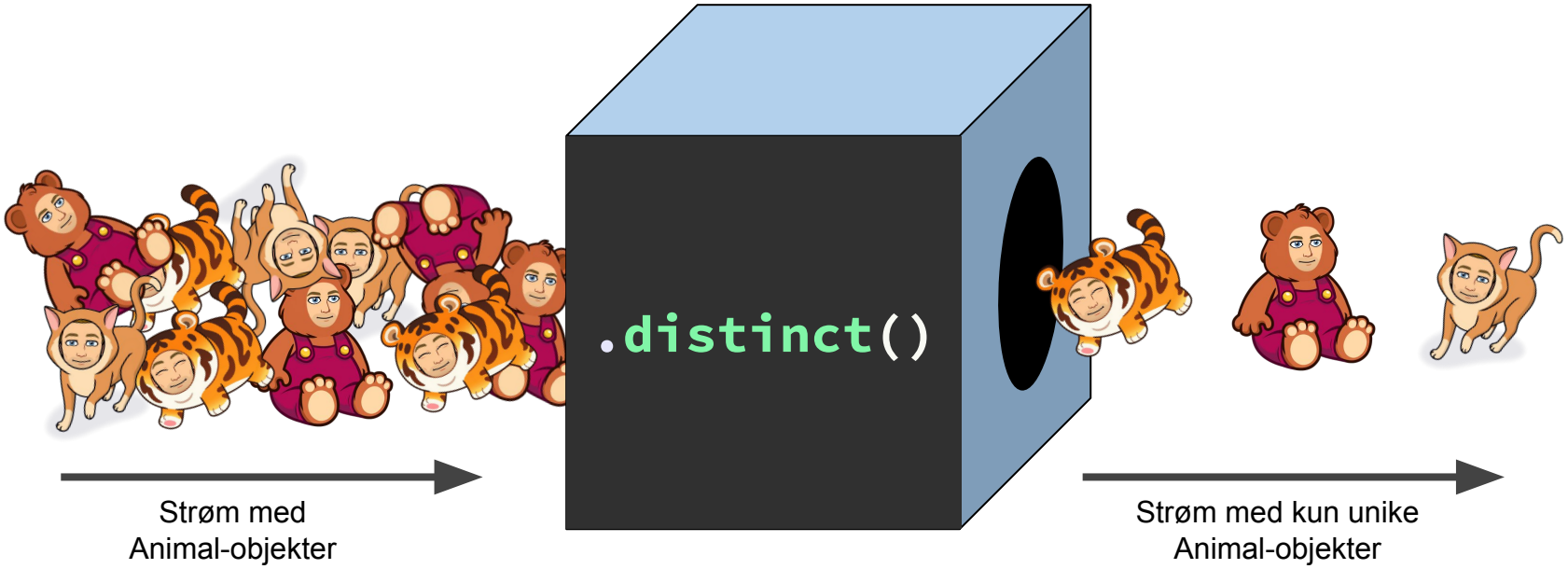
- Tar ikke inn noen argumenter.
- Returnerer en *strøm* med kun **distinkte** (ulike) elementer.

```
List<Person> distinct = new ArrayList<Person>();

for (Person p : persons) {
    if(!distinct.contains(p)) {
        distinct.add(p);
    }
}
System.out.println(ages);
```

```
System.out.println(persons.stream().distinct().toList());
```

`distinct()`-funksjonen:

Visualisering



NTNU

Nyttige metoder i Stream:

`toList()`

- Samler innholdet i *strømmen* til en **immutable** liste
  - Immutable: Innholdet i listen kan ikke endres
  - Dette er tilsvarende oppførsel som **`List.of()`**

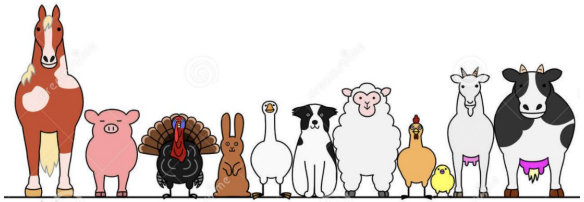
- Brukes som siste metode i en lenke med en eller flere Stream-metoder:

```
persons.stream().distinct().toList();
```

Oppgave 6

Vi lagde tidligere **Comparator**-objekter for å sammenligne to dyr basert på alderen deres. Vi ønsker nå å sortere en liste av dyr basert på alfabetisk rekkefølge av navnene deres.

I filen Oppgave6.txt finnes det en main-metode som lager en liste med dyr, sorterer disse på alder og skriver dem ut i sortert rekkefølge. Endre denne til å sortere dyrene i alfabetisk rekkefølge uten å endre på koden utenfor metoden. (Lambdauttrykk)

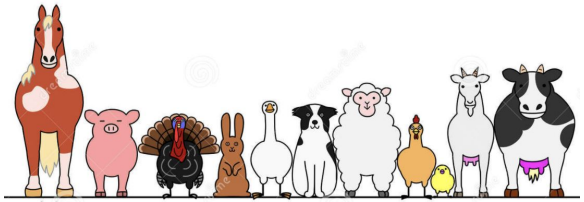


Oppgave 7

Vi ønsker nå muligheten for å finne navnene til alle dyrene på gården. Vi ønsker **ikke** at denne metoden gir ut **duplikate** navn flere ganger.

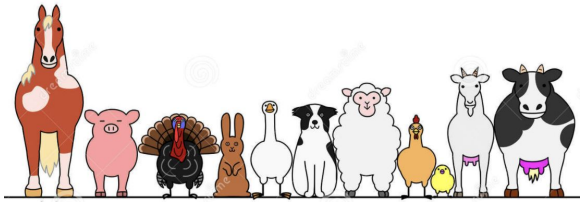
Lag en metode getAnimalNames i Farm-klassen som returnerer en liste med de unike navnene til dyrene på gården. Tips: Bruk streams, og se gjerne på map, distinct og toList

main-metoden i filen Oppgave7.txt kan kjøres for å teste implementasjonen din. Ved riktig implementasjon skal den skrive ut tre navn.



Oppgave 8 (Ekstra, hvis tid)

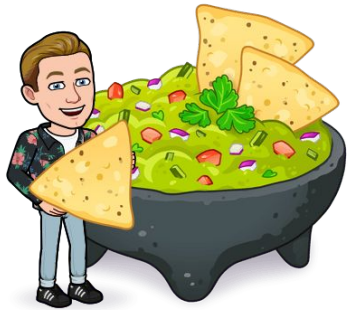
Implementer tilsvarende funksjonalitet for å iterere kun på Dog-objekter som i oppgave 5, men denne gangen ved bruk av streams og .filter()-funksjonen.



NTNU

Teorisnacks

Ekstra slides for de som vil lære litt mer om grensesnitt



Funksjoner som objekter

- Vi kan **strengt tatt** ikke deklarere rene funksjoner som objekt i Java, men vi kan opprette *objekter med kun én metode*, som blir så å si det samme i praksis. Dette er grunnen til at vi har **funksjonelle grensesnitt** med kun én metode.

- Lambda-syntaksen slår dermed sammen det å implementere grensesnitt og det å instansiere et klasse som et objekt i én operasjon.

- Dette er en litt ny måte å tenke på, men det bygger opp under ideen om at alt **alt i Java er objekter****

- Å definere funksjoner som objekt lar oss for eksempel bruke disse funksjonene som **inputargument** i andre **funksjoner/metoder** igjen eller noe vi kan returnere. Funksjoner kan dermed også tilordnes variabler

**Se bøkene *Object Thinking* av David West og *Clean Code* av Robert C. Martin for forskjellige synspunkt på dette

For spesielt interesserte:

`collect()`

- Tar inn et objekt av typen **`Collector`** (ikke Collection!) som samler innholdet til *strømmen* til flere forskjellige typer Collections.
- Brukes som siste metode i en lenke med en eller flere Stream-metoder.
- Det finnes mange Collectors, men noen av de mest vanlige er:
  - **`Collectors.toList()`**
  - **`Collectors.toMap()`**
  - Det finnes altså flere forskjellige metoder i **`Collectors`** som samler strømmen til forskjellige **`Collection`**-typer

```
persons.stream().distinct().collect(Collectors.toList());
```

Type casting

- Vi 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 gjennom 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;
```