

menti.com kode: 41 466

Beholdere og generiske klasser II

IN1010 uke 7

Onsdag 26. februar 2020

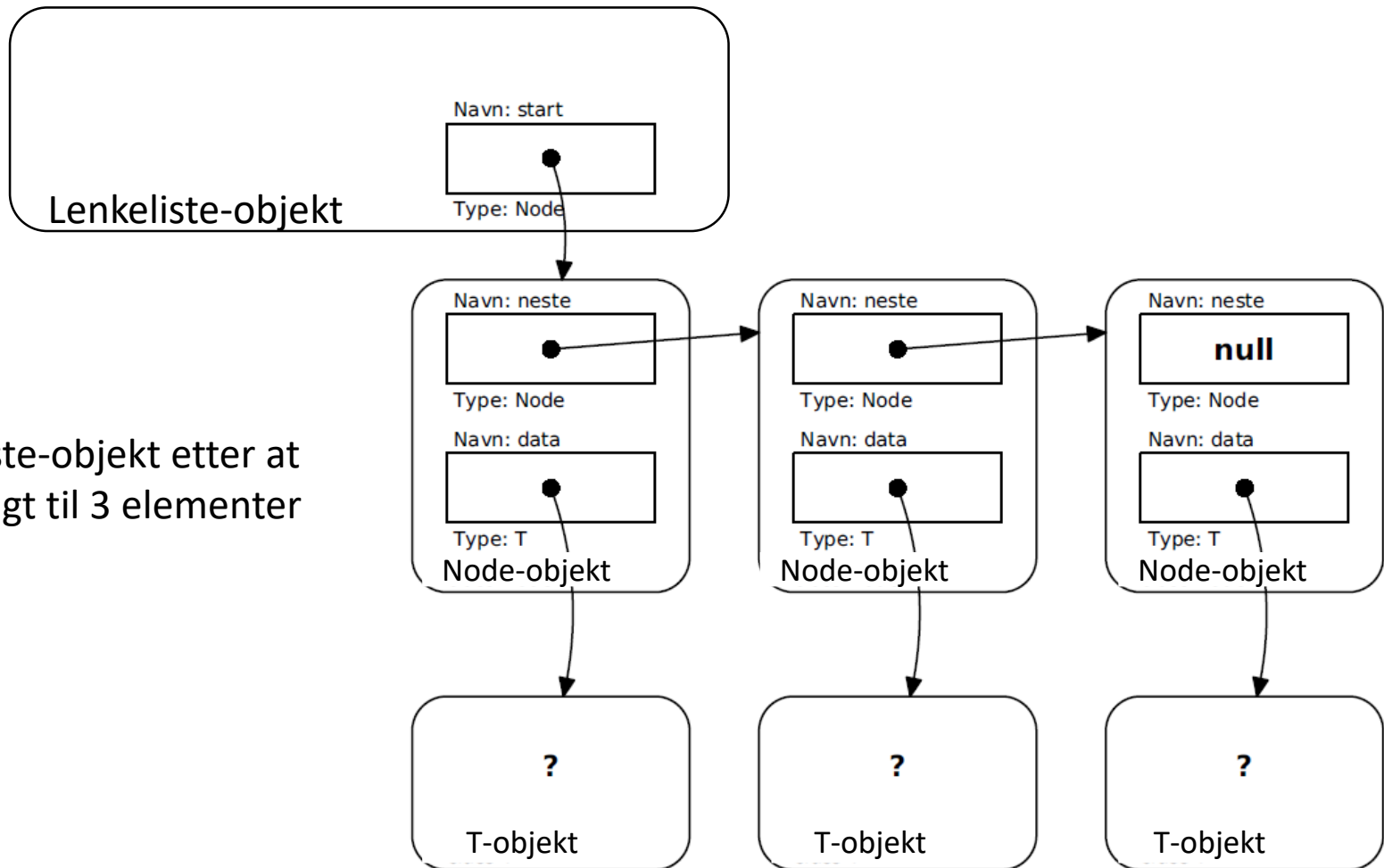
Beholdere og generiske klasser - II

- Implementasjon av lenkelister
 - Enveislister (som vi så på sist)
 - Toveislister (nytt)
 - Egen referanse til siste element i listen
- Varianter av lister:
 - stabel (stack, Last In First Out – LIFO)
 - kø (First In First Out – FIFO)
 - Prioritetskø
- Mer Java
 - Innpakking ("boxing")
 - Å sammenligne objekter (Comparable)
 - Å gå gjennom alle elementer i en beholder (Iterator)

Oblig 4: Arbeid i grupper

- ALL info om sammensetting og registrering av grupper ligger på en side lenket fra oblig4 på oblig-siden (kommer i dag/ nå).
- Opprette/ registrere gruppe på seminartimene frem til mandag
- Oppgaven legges ut neste uke til vanlig tid

Forrige uke: Klassen Lenkeliste



Lenkeliste-objekt etter at vi har lagt til 3 elementer

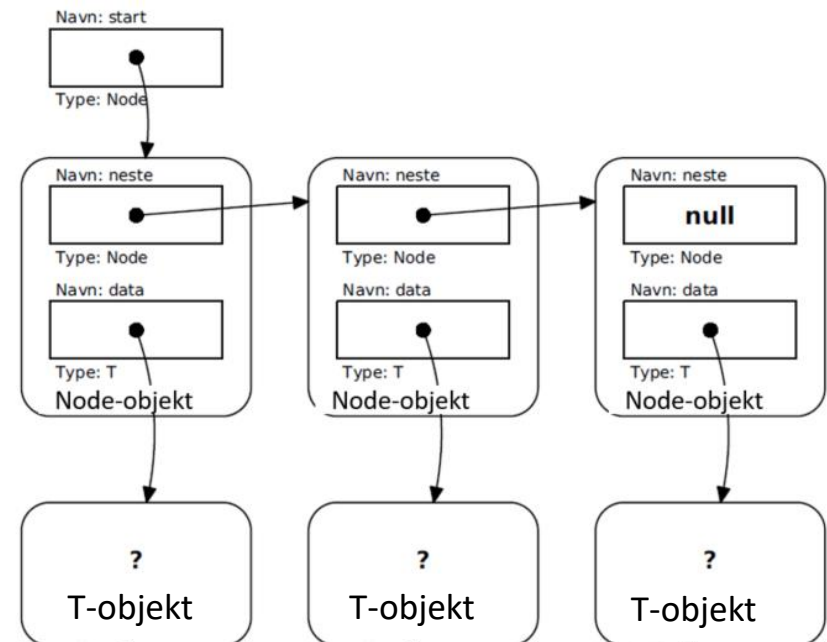
Hvordan hente et element?

```
public T get(int pos) {}
```

- Går gjennom liste, teller oss frem til rett plass

```
Node p = start;  
for (int i=0; i<pos; i++) {  
    p = p.neste;  
}
```

- NB: Hva skal vi returnere?

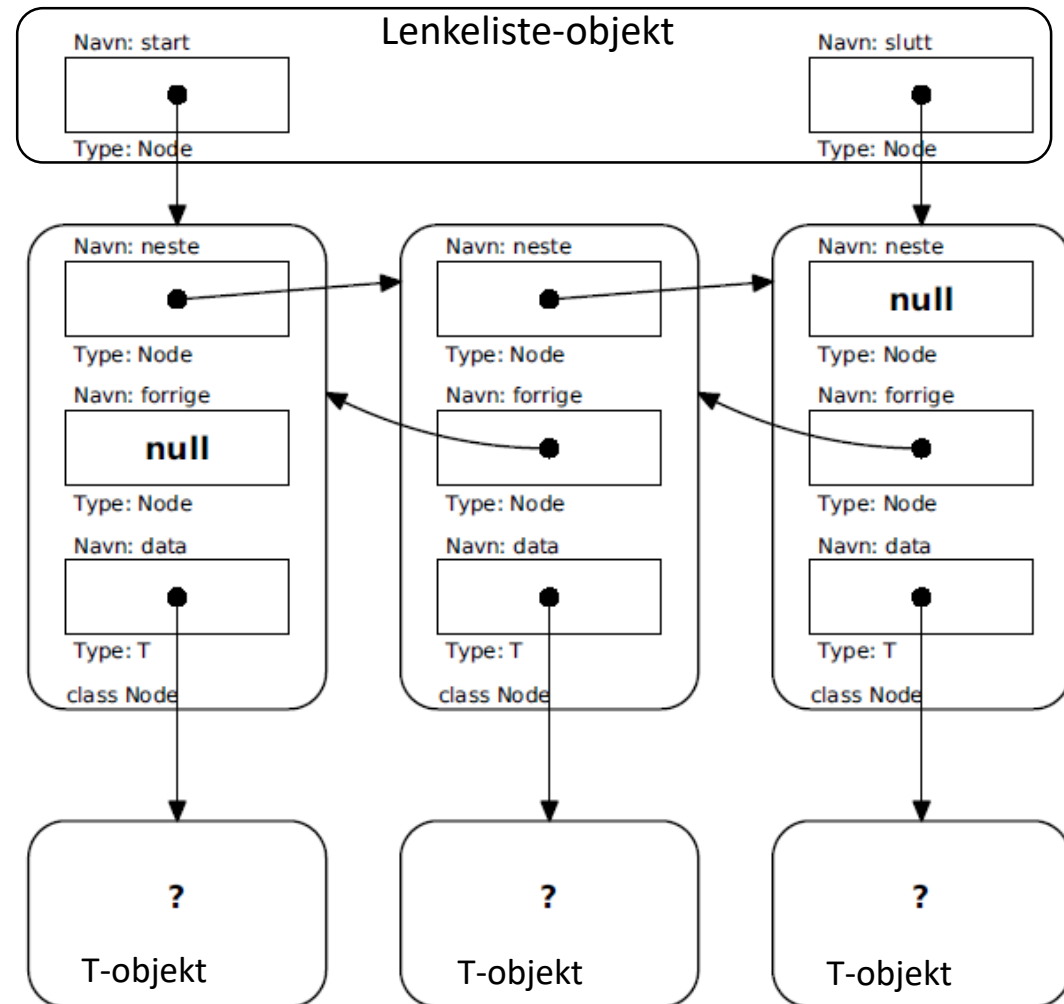


Fordeler og ulemper ved enkeltlenket liste

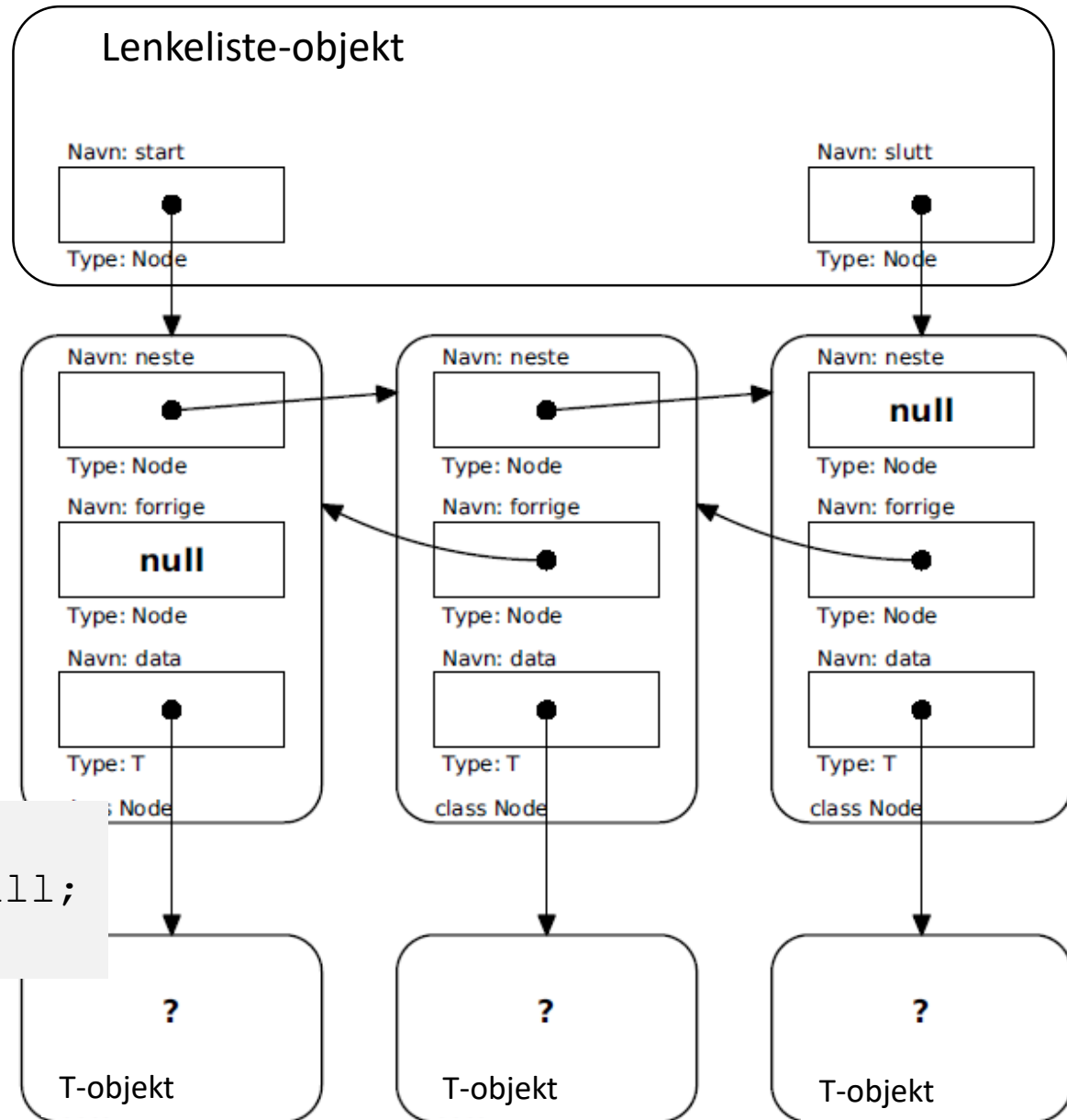
- + Fleksibel! Alltid plass til akkurat så mange elementer som trengs
- + (relativt) enkelt å programmere metodene
- Mye "nesting" for å jobbe seg frem til spesielt elementer som ligger langt ut i listen

Toveisliste

```
class Node {  
    Node neste = null;  
    Node forrige = null;  
    T data;  
    Node(T x) {  
        data = x;  
    }  
}  
private Node start=null;  
private Node slutt=null;
```



Toveisliste



- Slipper å lete oss frem til siste element ved innsetting, fjerning, endring
- Eks - fjerne siste element:

```
Node n = slutt;  
slutt.forrige.neste = null;  
slutt = slutt.forrige;
```


Valg av datastruktur for en liste

- Det er vanligvis fornuftig å ha en instansvariabel **slutt** også i en enveis liste
- Da kan man i hvert fall enkelt lese av/ endre det siste, og legge til nytt element bakerst
- Det kan være fornuftig å velge toveis liste når
 - man ofte skal finne et av de siste elementene i en liste
 - man ofte skal fjerne andre elementer enn det første
- Ellers greit å bruke enveis liste der både strukturen og (stort sett) programmeringen er enklere
- En instansvariabel for størrelse gjør at man ikke trenger traversere for å finne størrelsen

Elementer av primitive typer

- Klasseparametere kan kun representere referanse-typer!
- Hvordan kan vi lagre og organisere for eksempel heltall, boolske verdier eller tegn?
- En klasse som representerer et heltall

```
public class Heltall {  
    private int verdi;  
    public Heltall(int i) {  
        verdi = i;  
    }  
    public int intValue() {  
        return verdi;  
    }  
}
```

Innpakking av primitive typer

- Da kan vi lage en liste av heltall ved å «pakke dem inn» ett og ett i objekter (såkalt «boxing») og kan bruke listen som normalt:

```
Liste<Heltall> lx = new Lenkeliste<>();  
lx.add(new Heltall(12));  
int v = lx.get(0).intValue();
```

Java's "innpakkings-klasser"

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

```
intObj = new Integer(5);  
intObj = Integer.valueOf(5);  
heltall = intObj.intValue();
```

- ++ en rekke andre nyttige metoder for konvertering pluss konstanter for største/minste verdi

Automatisk inn- og utpakking

- Innpakking fungerer bra:

```
ArrayList<Integer> lx = new ArrayList<>();  
lx.add(Integer.valueOf(12));  
int v = lx.get(0).intValue();
```

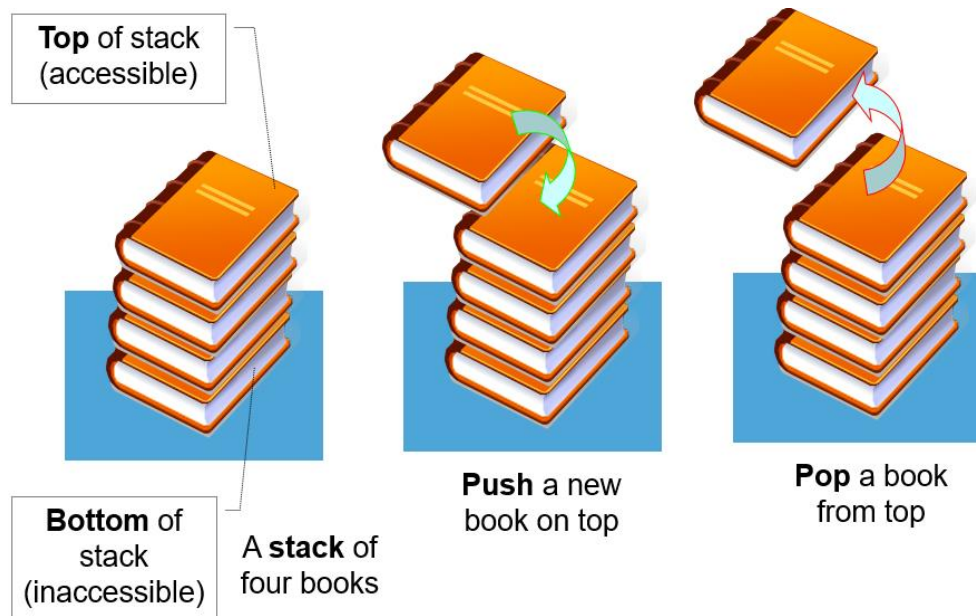
- ... men notasjonen er litt kronglete. Derfor tillater Java automatisk inn- og utpakking for standardklassene med kortformen:

```
ArrayList<Integer> lx = new ArrayList<>();  
lx.add(12);  
int v = lx.get(0);
```

Stabel («stack»)

Last in First Out – LIFO

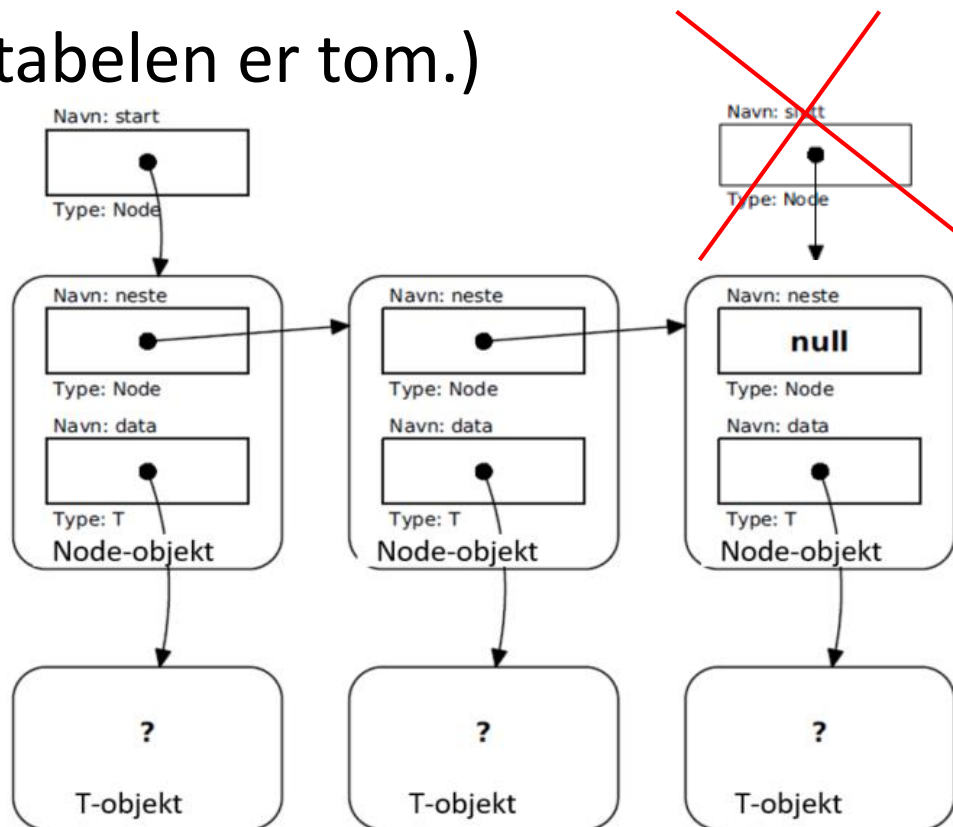
- En spesiell form for lister er stabler der vi kun legger nye elementer på toppen («push-er») og kun henter fra samme ende («pop-er»).
- Vi jobber hele tiden på toppen = i begynnelsen av listen



Legge til nytt element («push»)

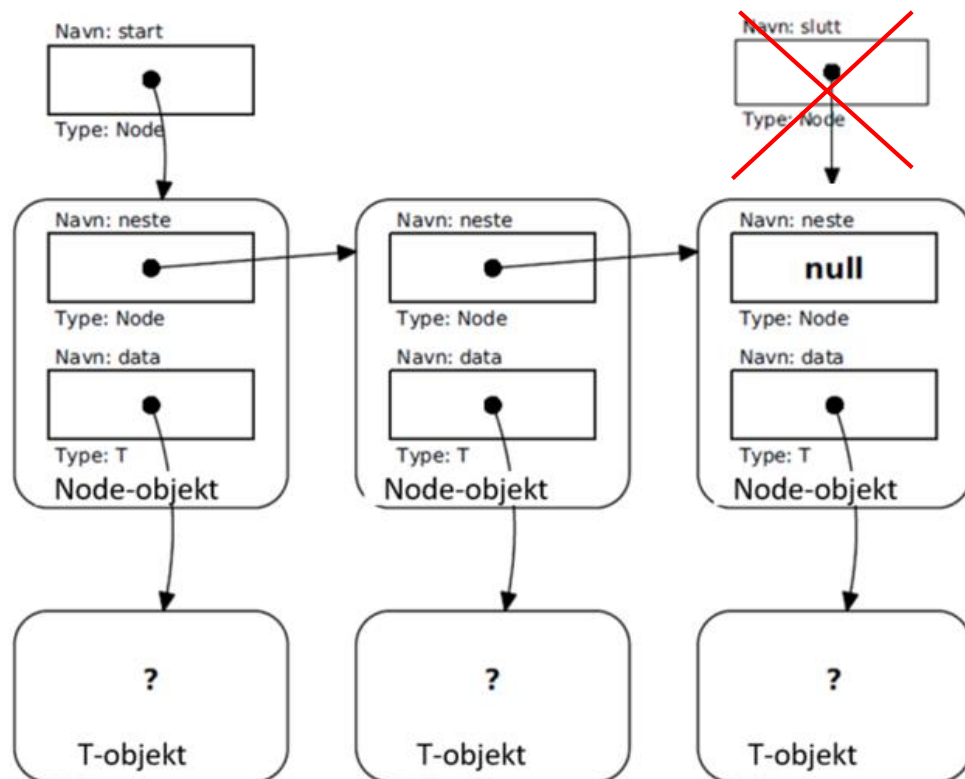
(Fungerer bra også om stabelen er tom.)

```
Node ny = new Node(v);  
ny.neste = start;  
start = ny;
```



Ta av element («pop»)

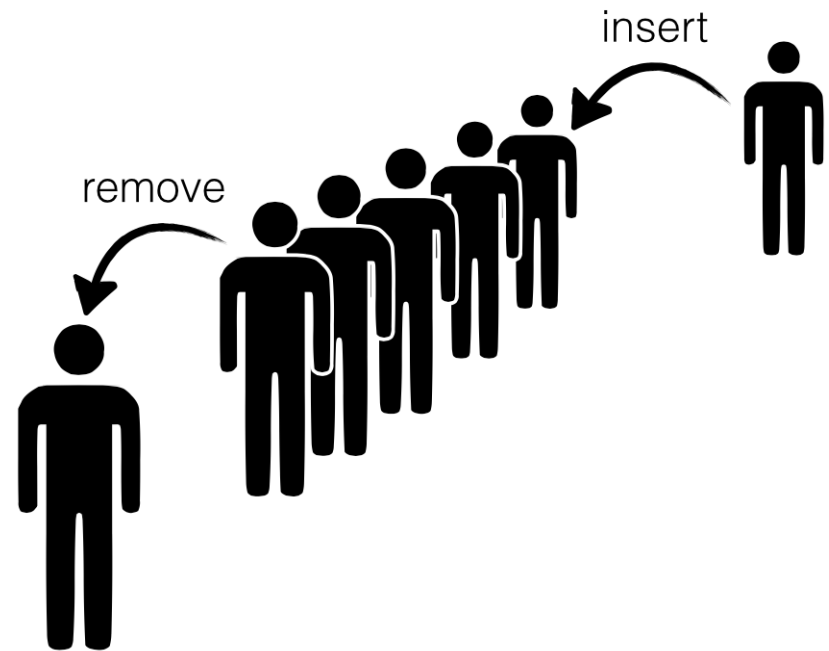
```
T svar = null;  
if (start != null) {  
    svar = start.data;  
    start = start.neste;  
}
```



Kø («queue»)

First in First Out - FIFO

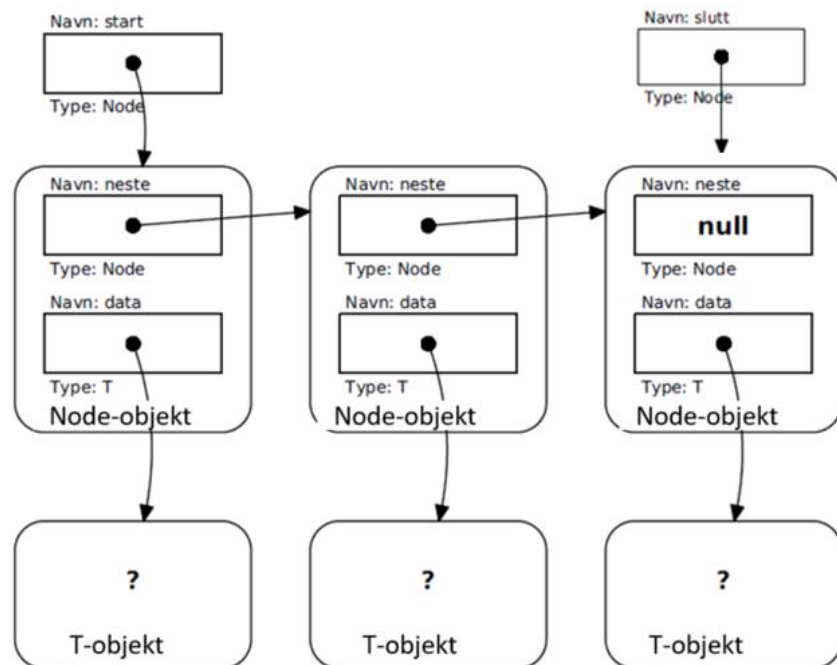
- En annen spesiell form for lister der vi alltid setter inn nye elementer bakerst og henter dem ut fra starten.



Inn i / ut av en kø

- Sette inn i kø:

```
Node ny = new Node(v);  
slutt.neste = ny;  
slutt = ny;
```



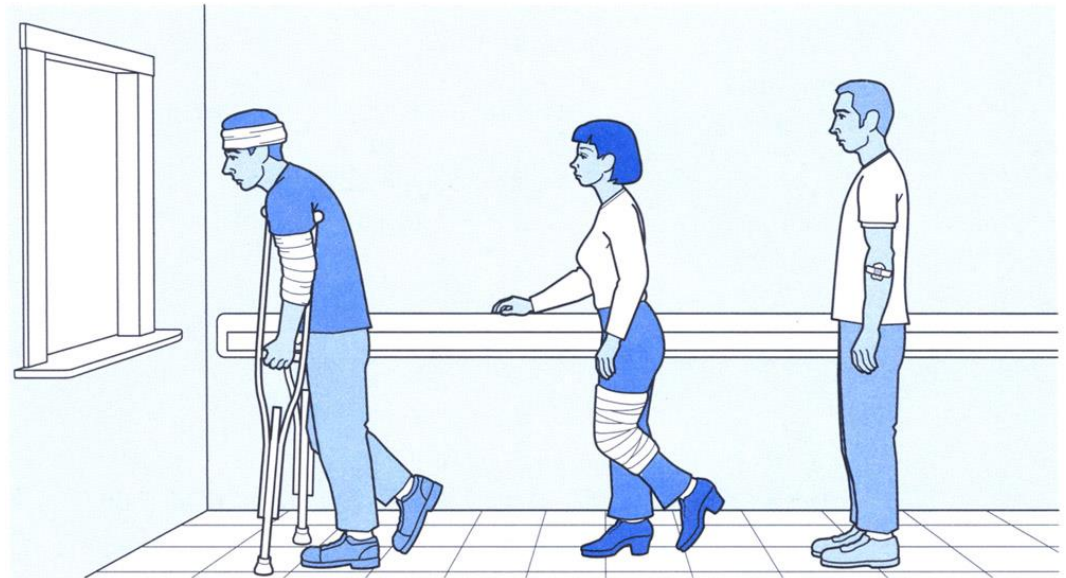
(Men hva om køen er tom når vi skal sette inn?)

- Hente ut av køen

- Akkurat som å pop-e en stabel

Prioritetskø («Priority queue»)

- I prioritetskøer haster noen elementer mer enn andre; de tas først uavhengig av hvor lenge de har stått i køen.



Implementere en prioritetskø

- Vil holde vår prioritetskø sortert ved at nye elementer settes på korrekt plass.
- Hvordan vet vi hva som er riktig sortering?
- Vi trenger en standardtest som forteller oss om prioritetsforholdet for to elementer p og q :
 - Er $p < q$?
 - Er $p = q$?
 - Er $p > q$?

Interface Comparable

- Java-biblioteket har et interface kalt Comparable som angir at noe er sammenlignbart. Det inneholder kun én metode

```
public interface Comparable<T> {  
    public int compareTo(T otherObj);  
}
```

- Ideen er at vårt objekt skal kunne sammenlignes med et vilkårlig annet objekt og returnere et heltall
 - < 0 hvis vårt objekt er *mindre* enn det andre
 - $= 0$ hvis vårt objekt er *likt* med det andre
 - > 0 hvis vårt objekt er *større* enn det andre

Eksempel på bruk av Comparable

```
public class Deltagerland implements Comparable<Deltagerland> {
    private String navn;
    private int antGull, antSoelv, antBronse;
    Deltagerland(String id, int g, int s, int b) {
        navn = id; antGull = g; antSoelv = s; antBronse = b; }
    @Override
    public int compareTo(Deltagerland a) {
        if (antGull < a.antGull) return -1;
        if (antGull > a.antGull) return 1;
        if (antSoelv < a.antSoelv) return -1;
        if (antSoelv > a.antSoelv) return 1;
        if (antBronse < a.antBronse) return -1;
        if (antBronse > a.antBronse) return 1;
        return 0;
    }
}
```

Tester program med Comparable

```
class TestMedaljer {  
    public static void main(String[] args) {  
        Deltagerland danmark = new Deltagerland("Danmark", 0, 0, 0),  
        finland = new Deltagerland("Finland", 1, 1, 4),  
        island = new Deltagerland("Island", 0, 0, 0),  
        norge = new Deltagerland("Norge", 14, 14, 11),  
        sverige = new Deltagerland("Sverige", 7, 6, 1);  
  
        System.out.println("Finland vs Sverige: " +  
            finland.compareTo(sverige));  
  
        System.out.println("Norge vs Sverige: " +  
            norge.compareTo(sverige));  
  
        System.out.println("Danmark vs Sverige: " +  
            danmark.compareTo(sverige));  
  
        System.out.println("Danmark vs Island: " +  
            danmark.compareTo(island));  
    }  
}
```

**Finland vs Sverige: -1
Norge vs Sverige: 1
Danmark vs Sverige: -1
Danmark vs Island: 0**

Sorterer vha Comparable

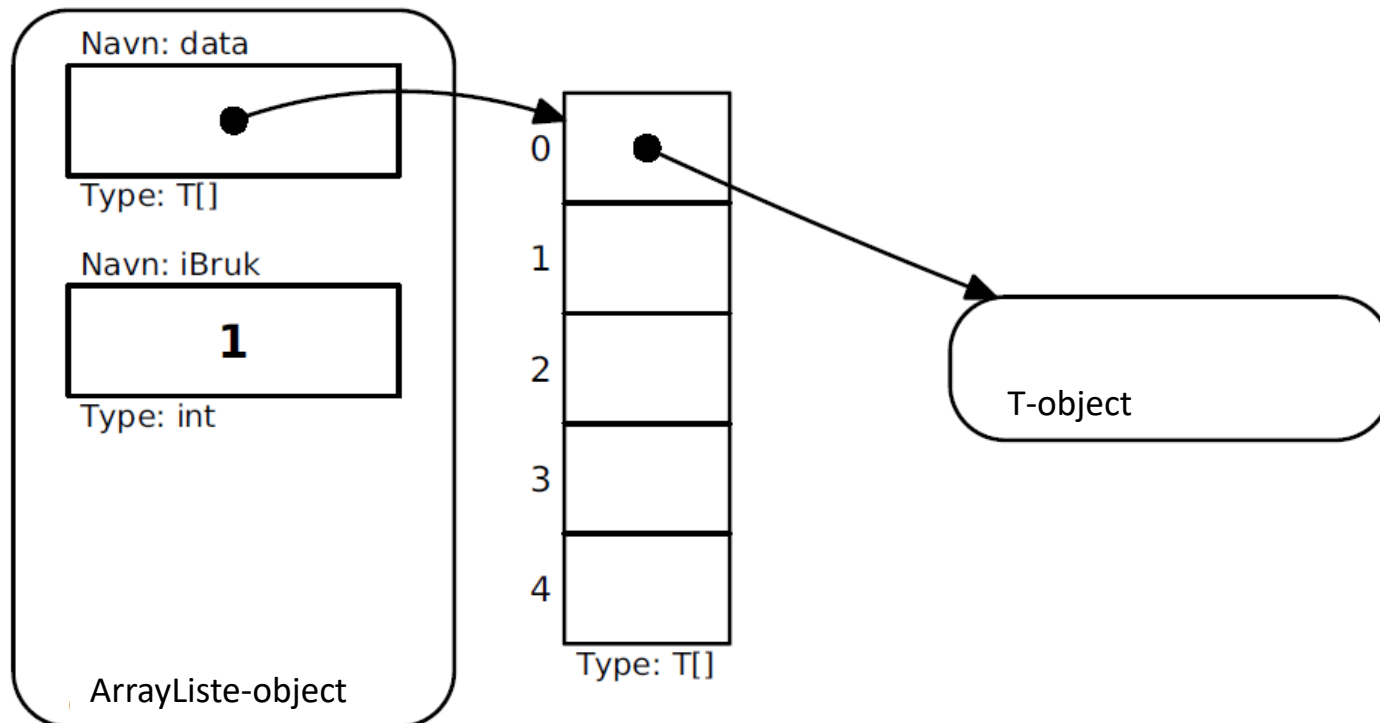
```
import java.util.Arrays;
class TestMedaljer2 {
    public static void main(String[] args) {
        Deltagerland[] land = {
            new Deltagerland("Danmark", 0, 0, 0),
            new Deltagerland("Finland", 1, 1, 4),
            new Deltagerland("Island", 0, 0, 0),
            new Deltagerland("Norge", 14, 14, 11),
            new Deltagerland("Sverige", 7, 6, 1) };

        Arrays.sort(land);
        for (int i = 0; i < land.length; i++)
            System.out.println(land[i].navn);
    }
}
```

Danmark
Island
Finland
Sverige
Norge

Implementasjon av prioritetskø

Vi kan lage prioritetskø som en spesialisering av Arrayliste.



Implementasjon av prioritetskø

- Vi kan lage prioritetskø som en spesialisering av Arrayliste.
- I ArrayListe satte vi inn nye elementer bakerst (eller på gitt posisjon) og tok ut etter posisjon med **get**
- Vi bestemmer oss for en minimal endring av ArrayListe grensesnittet, dvs:
 - endrer **add** til alltid å sette inn slik at førsteprioritet (lavest verdi!!) er først i listen
 - ved bruk av grensesnittet forutsetter vi dermed at når neste element skal behandles kalles **remove** med parameter `pos = 0`

Klasse som implementerer prioritetskø ved hjelp av Arrayliste

```
public class ArrayPrioKoe<T extends Comparable<T>>
    extends Arrayliste<T> {
    @Override
    public void add(T x) {
```

- Klassen **ArrayPrioKoe** er en subklasse av **Arrayliste**.
- Klasseparameteren **T** *må* implementere Comparable så vi får en sammenligning å sortere etter (og denne sammenligningen må sammenfalle med prioritet)
- Vi redefinerer kun metoden **add** siden innsettingen skal gjøres sortert.
- De øvrige metodene i **Arrayliste** kan være som de er.

add-metode for prioritetskø

Må identifisere spesialtilfeller:

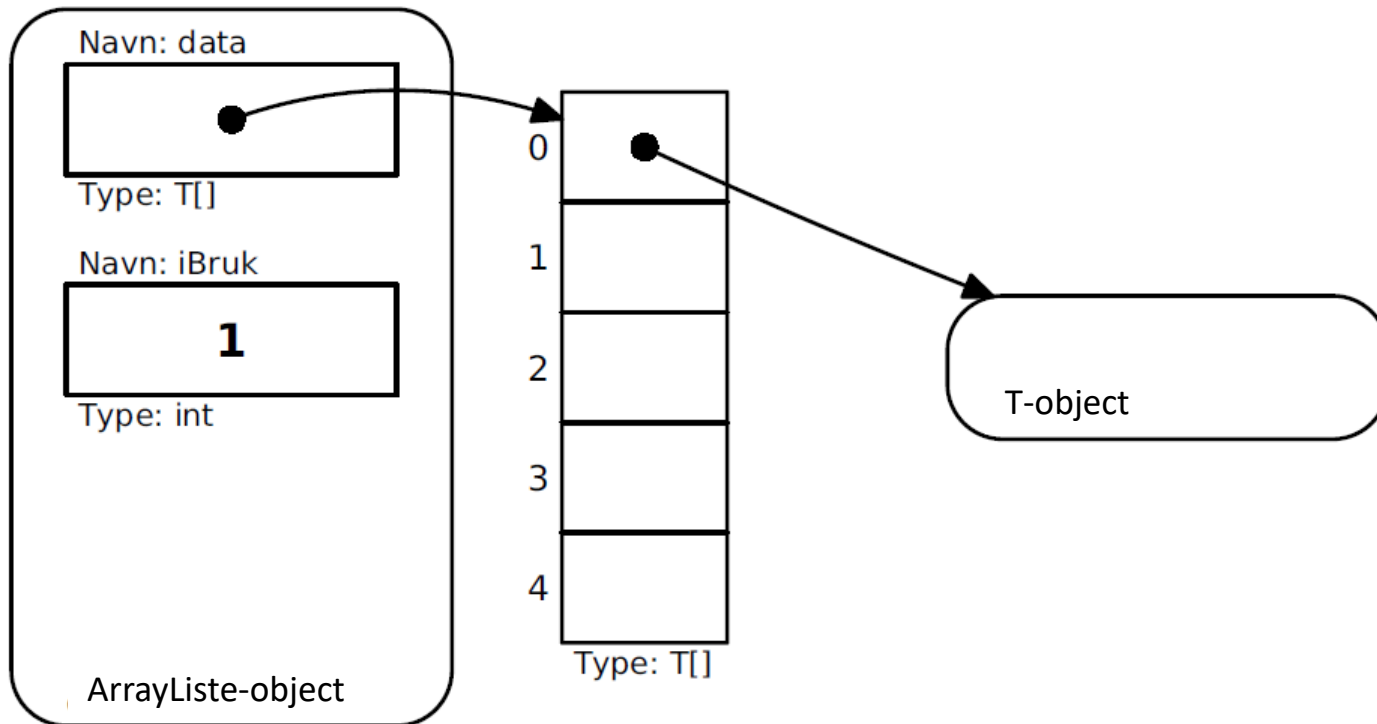
- Listen kan være tom fra før av.
- Hvis ikke, leter vi oss frem til første element i listen som er større enn det vi skal sette inn.
 - Flytter elementet og alle de resterende ett hakk lenger ut.
 - Setter inn det nye elementet på rett plass.
- Hvis alle elementene i listen er mindre eller lik det nye elementet, må det nye elementet settes bakerst.

add-metode for prioritetskø

```
public class ArrayPrioKoe<T extends Comparable<T>>
    extends Arrayliste<T> {

    @Override
    public void add(T x) {
        if (size() == 0) {
            // Listen er tom, så sett inn nytt element:
            super.add(x);
            return;
        }
    }
}
```

add i prioritetskø med Arrayliste



add (forts. med ikke-tom prioritetskø)

```
for (int i = 0; i < size(); i++) {
    if (get(i).compareTo(x) > 0) {
        // Vi har funnet et element som er større enn det nye.
        // Flytt det og etterfølgende elementer ett hakk opp.
        super.add(null); // Utvid arrayen (overskrives straks)
        for (int ix = size()-2; ix >= i; ix--)
            set(ix+1, get(ix));
        // Sett inn det nye elementet:
        set(i, x);
        return;
    }
}

// Det nye elementet er størst (ellers ville vi returnert)
// og skal inn bakerst:
super.add(x);

} // end of add
```

Tester add for prioritetskø

```
class TestPrio {  
    public static void main(String[] args) {  
        Liste<String> ap = new ArrayPrioKoe<>();  
        ap.add("Ellen"); ap.add("Stein"); ap.add("Siri");  
        ap.add("Dag"); ap.add("Anne"); ap.add("Irene");  
        for (int i = 0; i < ap.size(); i++)  
            System.out.println(ap.get(i));  
    }  
}
```

Anne
Dag
Ellen
Irene
Siri
Stein

- `compareTo` i `String`-klassen kalles implisitt i **add**

Prioritetskø: Forutsetninger/ valg

- Metoden **compareTo** baserer seg på en "natural ordering", *naturlig rekkefølge*
 - Hver klasse kan bare ha én compareTo metode
 - Hvis prioritet skal baseres på noe annet enn denne rekkefølgen (eller klassen som skal sammenlignes ikke implementerer compareTo) må vi bruke andre verktøy.
- Prioritet kan være lik eller motsatt av rekkefølge (her: "større" element plasseres lenger bak i køen)
- Det må være veldefinert om køen holdes sortert (som her) eller man må lete seg frem til rett element ved uttak

Å gå gjennom en Liste

- Vi kan gå gjennom en liste ved å hente elementene ett for ett med en indeks-basert for-løkke

```
Liste<String> lx = new ArrayListe<>();  
for (int i = 0; i < lx.size(); i++)  
    System.out.println(lx.get(i));
```

- Men Java har også en for-løkke som ligner den vi har brukt i Python

```
for (String s: lx)  
    System.out.println(s);
```

Javas interface **Iterator**

- En slik elegant gjennomgang er mulig med en iterator:

```
public interface Iterator<T> {  
    boolean hasNext();  
    T next();  
}
```

- En iterator holder orden på hvor langt vi er kommet i gjennomgangen og kan gi oss elementene ett for ett.

Implementasjon av iteratorer i Java

- for å kunne bruke for-each løkker på en beholder må beholderen kunne lage et iterator-objekt skreddersydd for beholder-klassen vår
- Vi trenger å gjøre følgende:
 - Spesifisere at grensesnittet til klassen vår implementerer interface **Iterable**:
Liste<T> extends Iterable<T>
 - Interface **Iterable** krever at beholder-klassen (ArrayListe) har en metode som returnerer en referanse til et **Iterator**-objekt "skreddersydd" for vår beholder
 - **Iterator** er et interface som krever implementasjon av metodene **next** og **hasNext** - dermed må vi også skrive en ny klasse som implementerer dette **Iterator**-grensesnittet for vår beholder **ArrayListe**.
 - Klassen som implementerer Iterator-grensesnittet for ArrayList kaller vi **ArrayListIterator**

Utvider interface Liste med Iterable

Interface Iterable gjør at vi kan bruke for-each løkker på klassen som implementerer det

Dette krever at klassen som implementerer det har en metode som leverer et objekt som tilbyr Iterator-grensesnitt (**next** og **hasNext**) for klassen.

```
interface Liste<T> extends Iterable <T> {  
    int size();  
    void add(T x);  
    void set(int pos, T x);  
    T get(int pos);  
    T remove(int pos);  
}
```

Liste.java

Interface Iterator

Dermed må klassen Arrayliste få

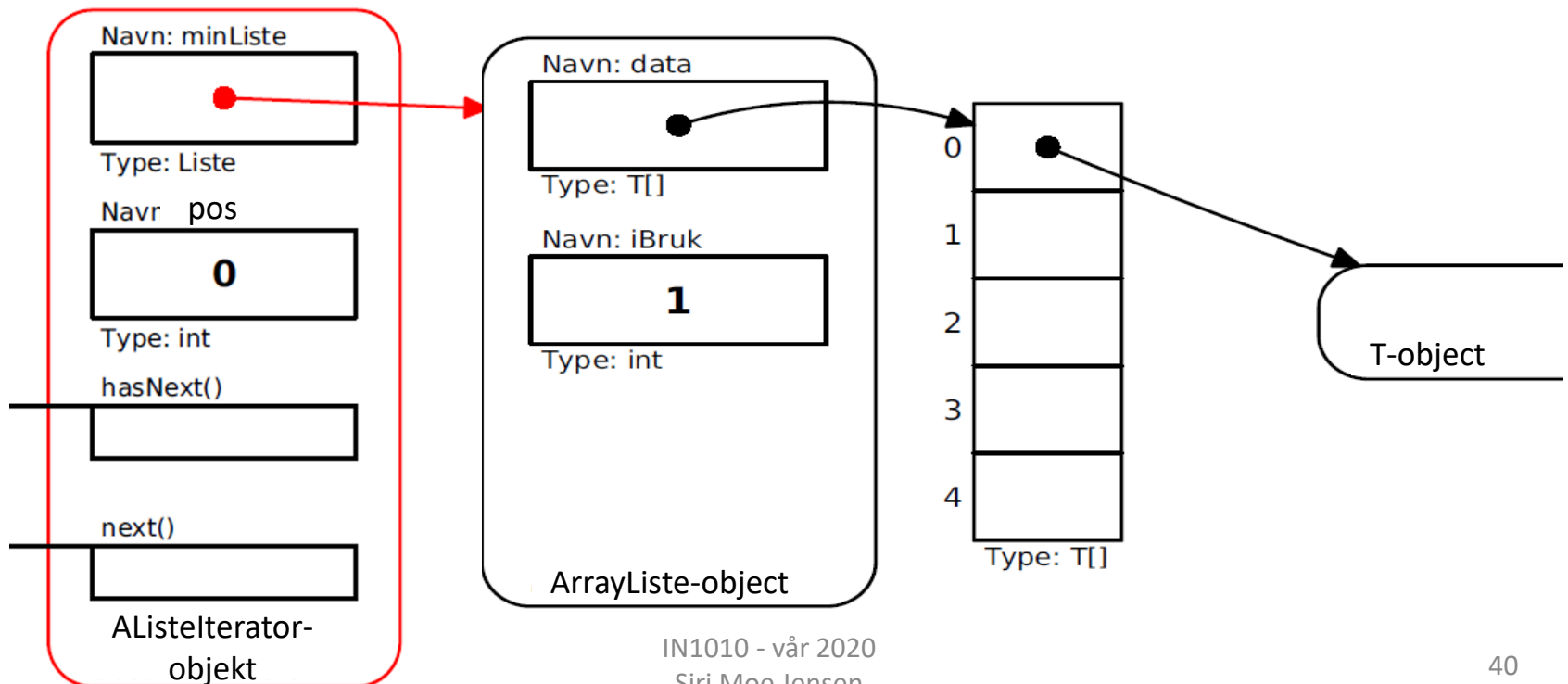
- en metode iterator (med *liten* forbokstav)
- som lager og returnerer et Iterator-objekt (med *stor* forbokstav)

```
class Arrayliste<T> implements Liste<T> {  
    @SuppressWarnings("unchecked")  
    private T[] data = (T[])new Object[10];  
    private int iBruk = 0;  
  
    public Iterator<T> iterator() {  
        return new AListeIterator<T>(this);  
    }  
  
    // Resten av metodene fra Liste-interfacet  
}
```

Arrayliste.java

Interface Iterator

- Iterator-interfacet krever implementering av **hasNext** og **next** for Arrayliste beholdere
- Da trenger vi en teller og en referanse til et Arrayliste objekt
- Implementerer dette i en klasse AListeliterator<T>



Interface Iterator

```
import java.util.Iterator;
class AListeIterator<T> implements Iterator<T> {
    private Liste<T> minAL;
    private int pos;

    public AListeIterator(Arrayliste<T> nyAL) {
        minAL = nyAL;
        pos = 0; }
    public T next() {
        return minAL.get(pos++); }
    public boolean hasNext() {
        return (pos < minAL.size()); }
}
```


Oppsummering

- Alternative datastrukturer for lenkelister
- Ulike grensesnitt for og implementasjon av lister
 - stabel
 - kø
 - prioritetskø
- Mer Java
 - Innpakking ("boxing")
 - Å sammenligne objekter (Comparable)
 - Å gå gjennom alle elementer i en beholder (Iterator)
- Neste uke: Rekursjon med Eyvind Wersted Axelsen