Beholdere 2

- IN1010
- Uke 7 (IN1010-tidsregning)
- Tirsdag 3. mars 2026

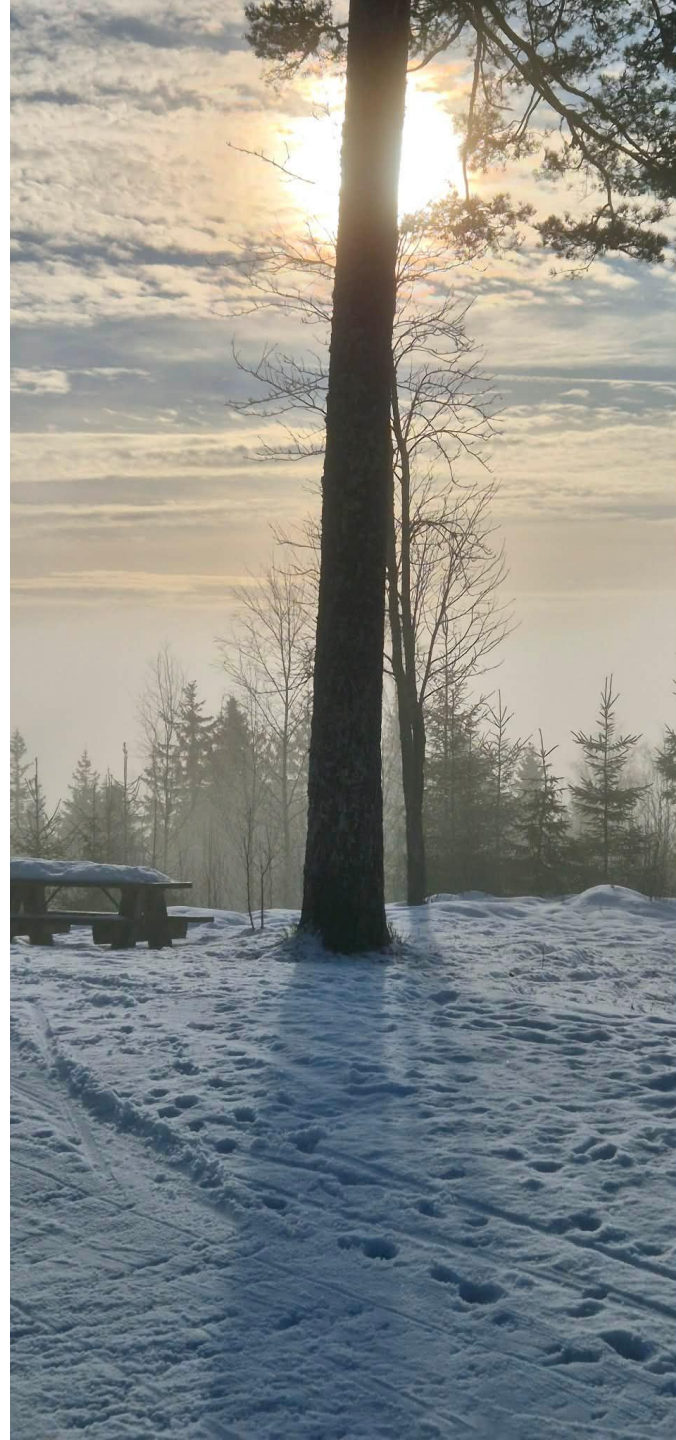


Forrige uke

- Beholdere, typehierarkier, Java Collections Framework
- Lister og et listegrensesnitt implementert med
  - Dynamisk array (referanser i et array)
  - Lenket liste (referanser fra sammenlenkede node-objekter)
- Testing av løsningene, feilhåndtering og unntak (Exceptions)
- Typeparametere og generiske typer
  - Deklarasjon av generiske grensesnitt og klasser
  - Instansiering og bruk av generiske klasser

Dagens forelesning

- Mer om datastrukturer for lister
- Nytt interface Liste<T>
  - Generiske typer, implementasjon av Liste<T> (mye repetisjon)
- Varianter av lister (ulik semantikk)
  - Stabel (stack)
  - Kø
  - Prioritetskø
- Mer Java
  - Innpakking ("boxing") av primitive typer
  - Å sammenligne objekter (Comparable)
  - Å gå gjennom alle elementer i en beholder (Iterator)

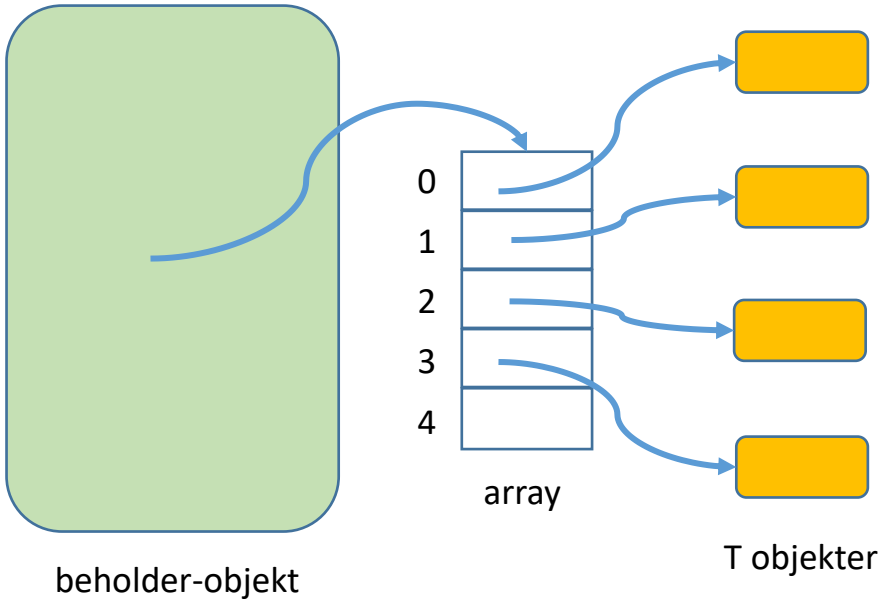


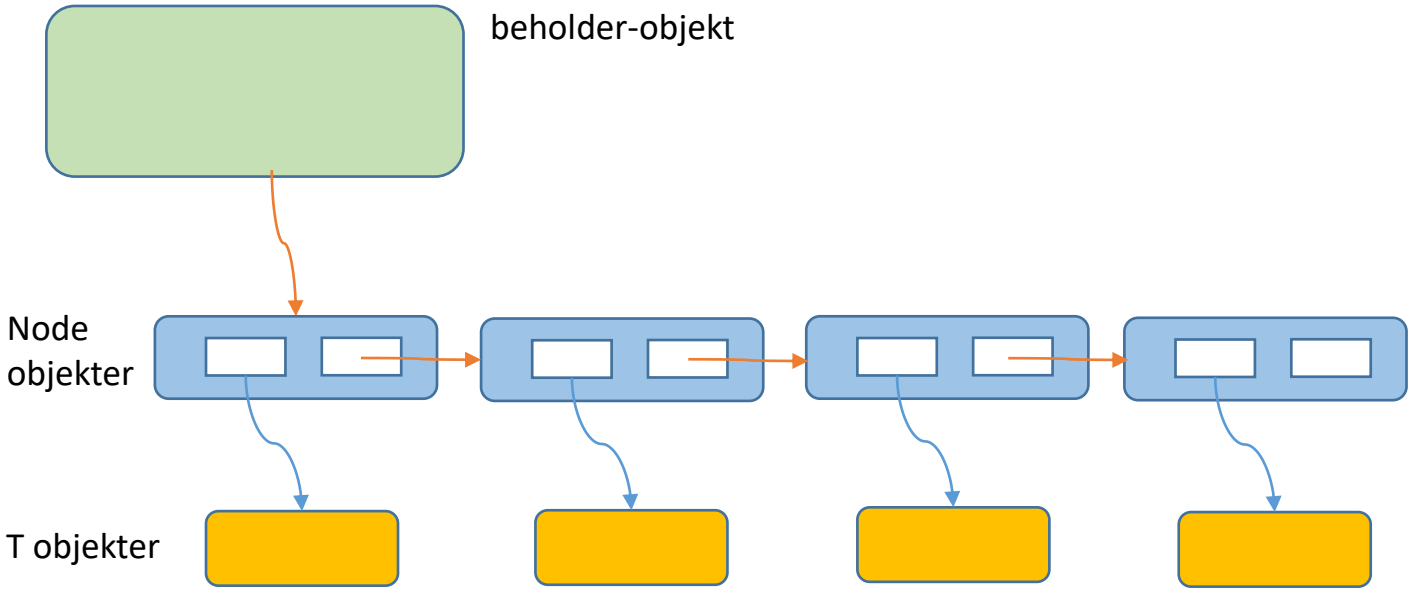
Hva ligger på semestersiden?

Lenkes fra timeplanen:

- Lysarkene
- Opptak
- Kode
  - Hele koden for klassen `Arrayliste` med tilbehør
  - Testprogrammer

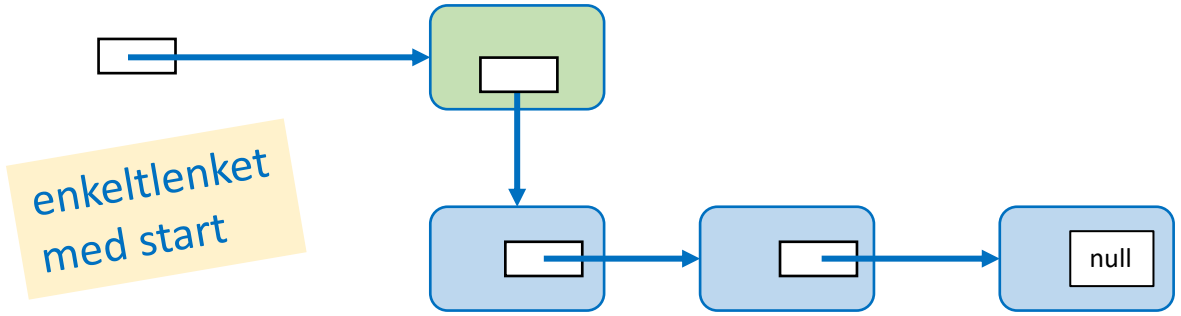
Lister implementert med array eller lenket liste (forenklet figur)

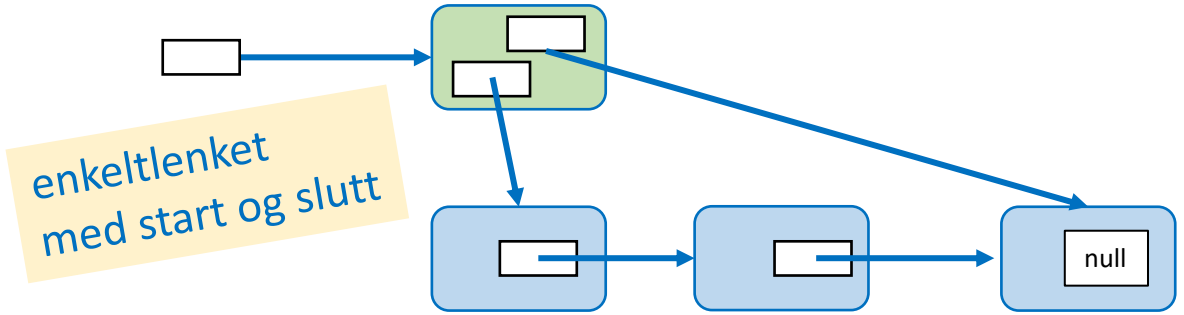


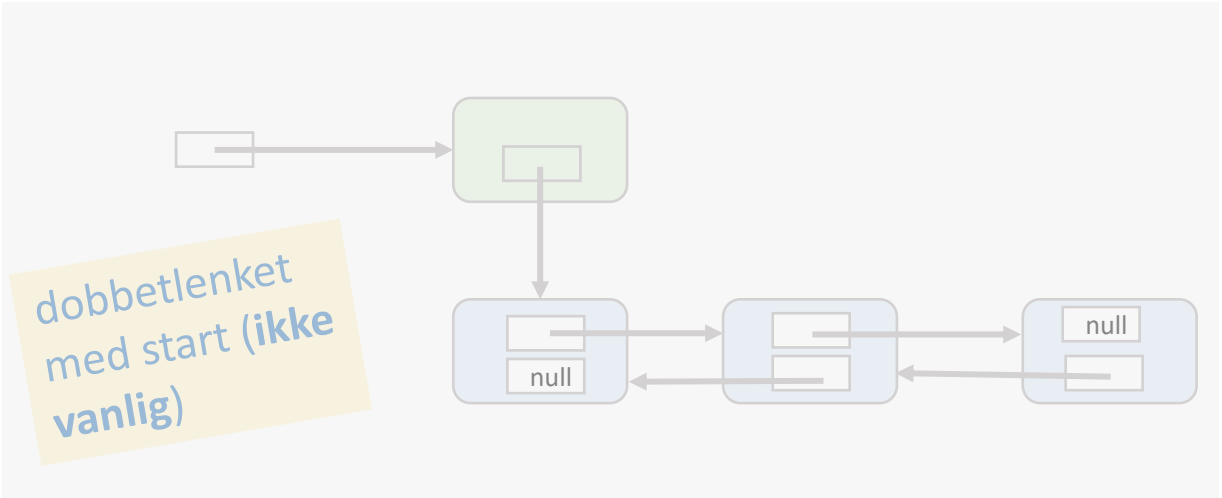


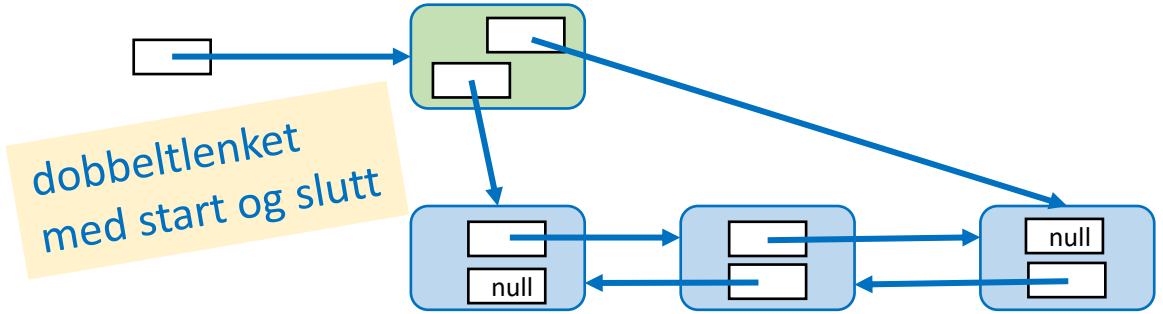
Lenkelister: Alternative datastrukturer

+ med eller uten instansvariabel **for størrelse**









Valg av datastruktur for en liste-beholder

- Både effektivitet ved kjøring, og effektivitet (og kvalitet) i utvikling, kan være relevante hensyn
- Grensesnitt og bruk vil påvirke effektiviteten og hva som er mest avgjørende

- Dynamiske arrayer som Java's `ArrayList` er raske
  - .. men noen innsettinger vil gå tregt

- Lenkete lister har alltid plass til akkurat riktig antall elementer
  - .. men er litt mer omstendelige å programmere og ofte litt tregere

Valg av variant for lenkete lister

- Det er vanligvis fornuftig å ha en instansvariabel *siste* i lenkelister
  - Enkelt å lese av/ endre det siste, og legge til nytt element bakerst
- En instansvariabel for størrelse tar liten plass og gjør at man ikke trenger traversere hver gang for å finne størrelsen

- Det kan være fornuftig å velge toveis lenkeliste om
  - man man trenger å traversere listen i begge retninger, eller
  - man ofte skal fjerne andre elementer enn det første

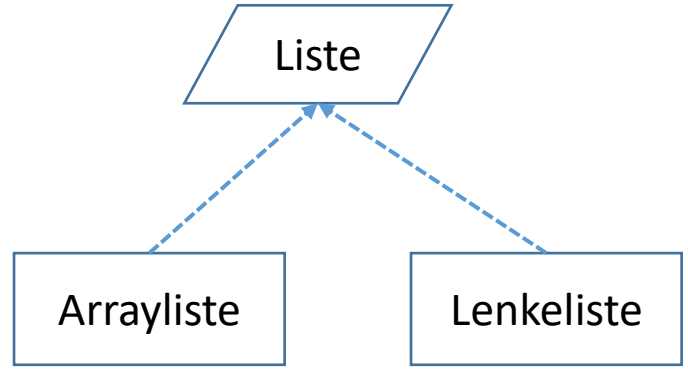
- Ellers greit å bruke enveis liste - strukturen og (stort sett) programmeringen blir enklere

Et nytt interface Liste

Litt repetisjon om generiske typer, et nytt interface Liste som viser noen alternativer til forrige ukes implementasjoner

Nytt typehierarki, interface og nye klasser

- Et annet `Liste` grensesnitt enn det vi så på forrige uke
- Tilbyr operasjoner på *posisjon* i listen (`set, get, remove`)
- Dessuten `size,` og `add`




```
interface Liste {
    int size();             //returner antall elementer i listen
    void add(??? x);        // legg til nytt element sist i listen
    void set(int pos, ??? x); // erstatt element i posisjon pos med x
    ??? get(int pos);       // returner referanse til element i pos
    ??? remove(int pos);    // fjern og returner element i pos
}
```

0 > pos <size-1

- Hvilken type skal vi angi ved innsetting og uthenting av elementer?

Et generisk Liste-interface (grensesnitt)

- Grensesnittet kan ha en typeparameter som representerer typen til elementene (mer presist: typen til pekerne til element-objektene) i listen.

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

Generiske klasser

- Som grensesnitt kan klasser i Java kan ha parametere som angir en (eller flere) *typer* som skal brukes i klassen. Dette kaller vi *generiske klasser* med *klasseparametere*

- Ved *instansiering* av en generisk klasse (opprettelse av et objekt) angir vi argument (aktuell parameter) for typeparameteren

- Brukes f.eks. i Java ArrayList:

```
ArrayList<String> minListe = new ArrayList<String>();
```

- Her er typen angitt implisitt:

```
ArrayList<String> minListe = new ArrayList<>();
```

- Java HashMap har to typeparametere: Man angir typen til nøklene og til verdiene

```
HashMap<String, Student> hm = new HashMap<>();
```

Deklarasjon av generiske klasser

- Klasser som implementerer vårt nye, generiske `Liste` grensesnitt

```
class Lenkeliste<T> implements Liste<T> {
    protected class Node {
        T data;
        Node neste = null;
        Node (T x) {
            data = x;
        }
    }

    protected int size = 0;
    protected Node første = null;
    // Metoder
}
```

```
class Arrayliste<T> implements Liste<T>{
    // Datastruktur og metoder
}
```

Eksempler på litt andre valg enn i forrige uke - vi ser nærmere på disse løsningene om litt

Mer om typeparametere

- Navnekonvesjon: Formell typeparameter angis med én stor bokstav
- Fra Java dokumentasjonen:
  - T - Type
  - S,U,V etc. - 2nd, 3rd, 4th types
  - E - Element (vanlig i Java Collections Framework)
  - K - Key
  - V - Value
- Bruk av typeparametere i Java: *Generics*

- En generisk type kan *begrenses*: Vi stiller krav til typen som kan brukes

```
// T must extend or implement Comparable<T>
class Node<E extends Comparable<E>> { ... }
```

Dataelementer av primitive typer

- Typeparametere kan kun representere referanse-typer!
- Hvordan kan vi lagre og organisere for eksempel heltall eller boolske verdier?
- Pakker verdien inn i et objekt av passende klasse: `Integer`, `Boolean` => *Boxing*
- Java hjelper oss med automatisk inn- og utpakking for standardklassene:

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

Generisk lenkeliste

Implementerer det nye `Liste` grensesnittet – repetisjon med andre løsninger

Datastruktur for ny lenkeliste

- Det finnes mange ulike liste-grensesnitt og måter å implementere dem med lenkede lister.
- Det er nyttig å se og trene på ulike varianter av datastruktur og andre implementasjonsvalg.

```
class Lenkeliste<T> implements Liste<T> {
    protected class Node {
        T data;
        Node neste = null;
        Node (T x) {
            data = x;
        }
    }

    protected int size = 0;
    protected Node første = null;
    // Metoder
}
```

Her er datastrukturen `protected` (ikke `private`). Da kan eventuelle subklasser aksessere den.

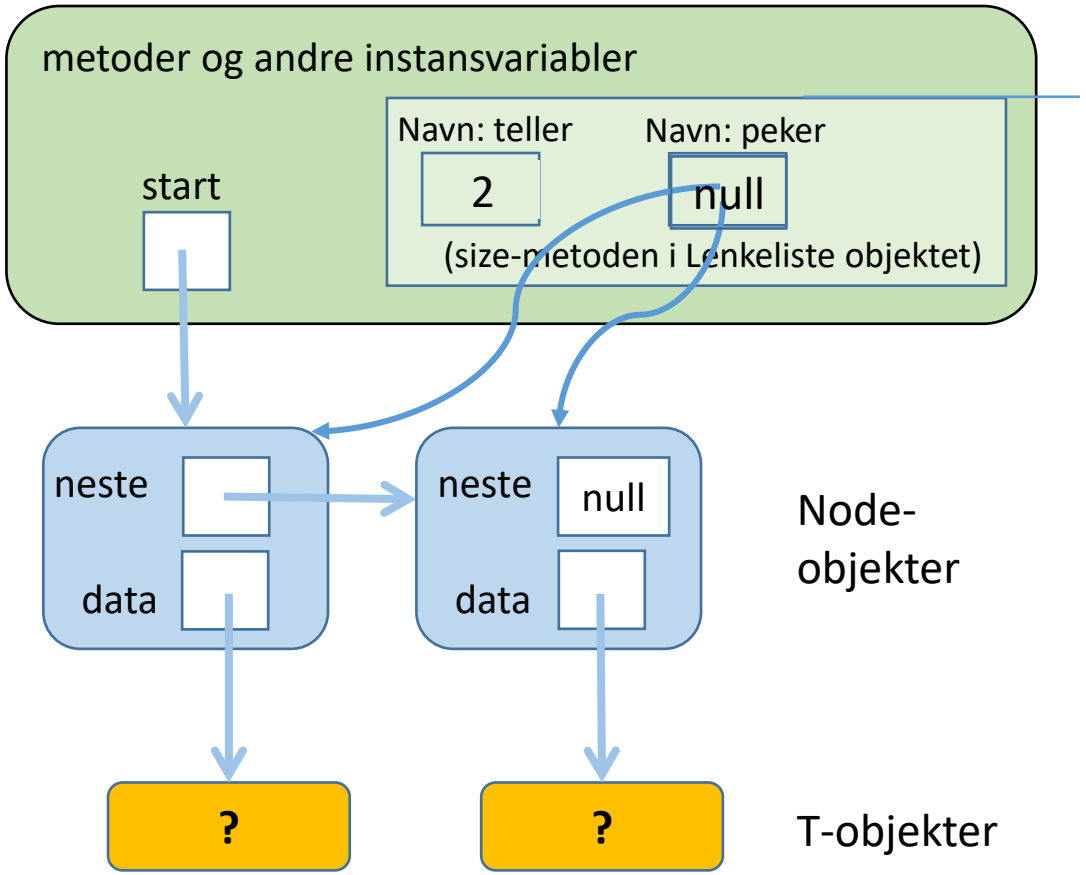
I denne implementasjonen bruker vi bare én parameter i Node-konstruktøren, og setter nestepekeren etter opprettelsen av ny Node og lett oss frem til rett plass. Det passer bedre når vi skal kunne sette inn på vilkårlig posisjon i listen.

Repetisjon (animasjon):

Hva gjør egentlig `peker=peker.neste`?

- `size()` som eksempel

```
public int size() {
    int teller = 0;
    Node peker = start;
    while (peker != null) {
        teller++;
        peker = peker.neste;
    }
    return teller;
}
```



Andre metoder i klassen `Lenkeliste`

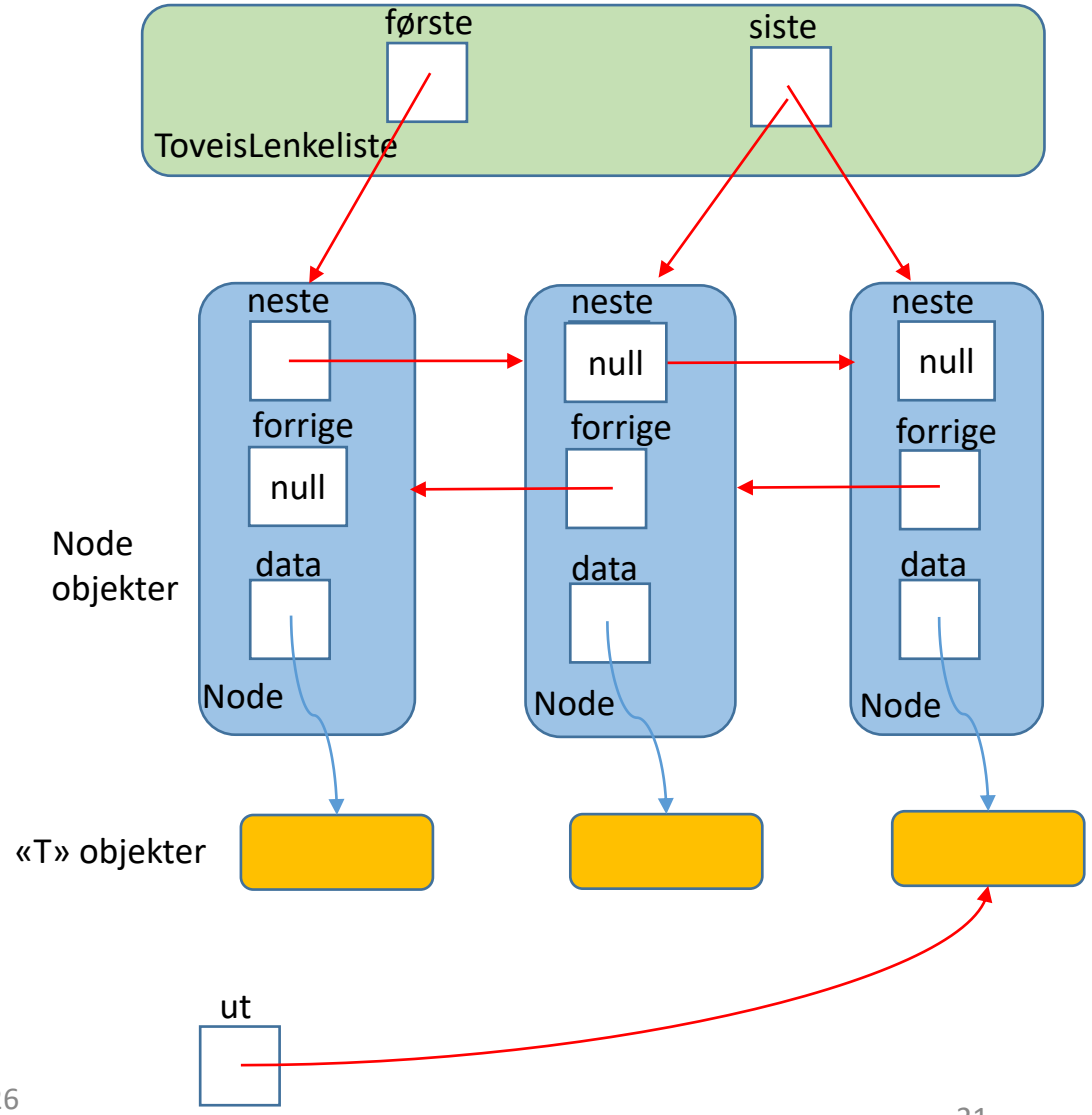
- Følger stort sett fremgangsmåtene i `EnkeltLenketStrengliste` fra forrige uke
- Må tilpasses:
  - Generisk beholder, håndterer ikke bare strenger
  - Implementerer det nye Liste-grensesnittet, dvs litt andre metoder
  - Jobber med andre instansvariabler

Toveis lenkeliste: Fjerne siste (animasjon)

- Skal fjerne *og* returnere referansen
- Må ta vare på den før vi fjerner!

```
T ut = siste.data;
siste = siste.forrige;
siste.neste = null;
return ut;
```

- Hvilke mulige spesialtilfeller må vi ta hensyn til her?
  - Tips: Se etter bruk av ‘.’ (risikerer NullPointerException)



Generisk Arrayliste

Implementerer det generiske Liste grensesnittet som dynamisk array

Datastruktur for generisk Arrayliste – og en mangel i Java

```
public class Arrayliste<T> implements Liste<T> {
    protected T[] data = new T[5];
```

```
Siri>javac *.java
Arrayliste.java:3: error: generic array creation
    protected T[] data =  new T[5];
                          ^
1 error
```

- Fungerer ikke?!

⇒ Java håndterer ikke arrayer med typeparametere

- Kan ha generiske klasser og interface – men *ikke generiske arrayer*

En "fix" – som fungerer

```
public class Arrayliste<T> implements Liste<T> {
    protected T[] data = (T[]) new Object[5];
```

Object[] som type for arrayen

NB: caster referansen til T[]

```
Siri>javac Arrayliste.java
Note: Arrayliste.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Advarsel fra Java: Objektene i arrayen har kanskje ikke T-egenskaper

Ignorerer advarselen fordi:

- alle metodene krever T-objekter i grensesnittet
- inne i beholderen bruker vi ingen av T-egenskapene

```
    @SuppressWarnings("unchecked")
    private T[] data = (T[]) new Object[5];
```

Datastruktur for Arrayliste klasse

```
public class Arrayliste<T> implements Liste<T> {
    protected T[] data = (T[]) new Object[5];
    protected int iBruk = 0;
```

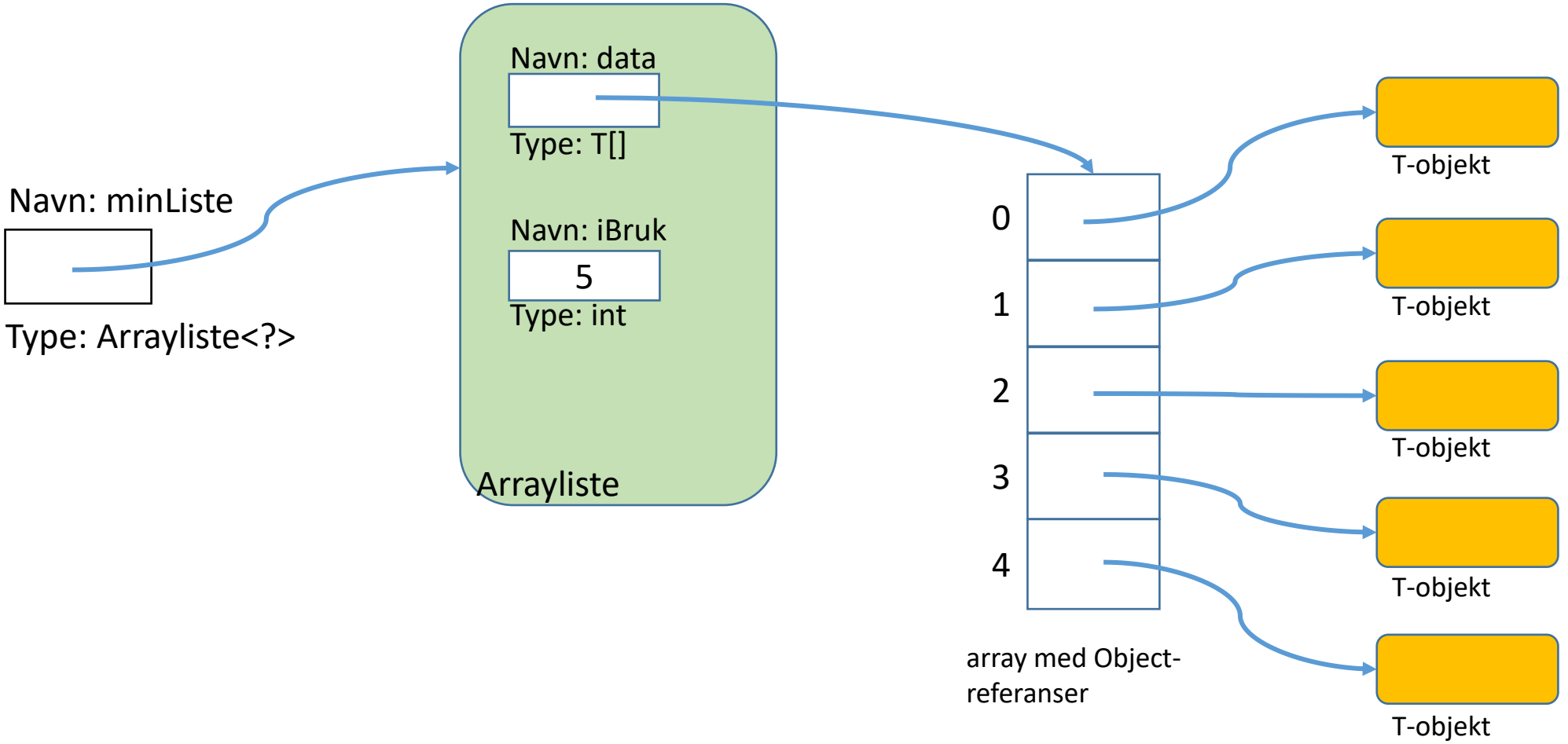
Samme prinsipper som forrige uke (men et annet grensesnitt, og generisk)

- Et array til å lagre elementene i
- Vi bruker bare så mange plasser i arrayet vi trenger
  => `iBruk` forteller hvor mange plasser som er i bruk
  OG hva som er neste ledige plass (hvis ikke arrayet er fullt)
- Trenger vi mer plass, erstatter vi arrayet med et nytt (f eks dobbelt så stort)

Arrayliste objekt

(her med 5 elementer)

Det finnes ikke T-objekter! Under kjøring vil de ha en annen, eksisterende type (for eksempel String) som vi opprettet listen med



Klassen Arrayliste (animasjon): size, set, get

```java
public class Arrayliste<T> implements Liste<T> {
    @SuppressWarnings("unchecked")
    protected T[] data = (T[]) new Object[5];
    protected int iBruk = 0;

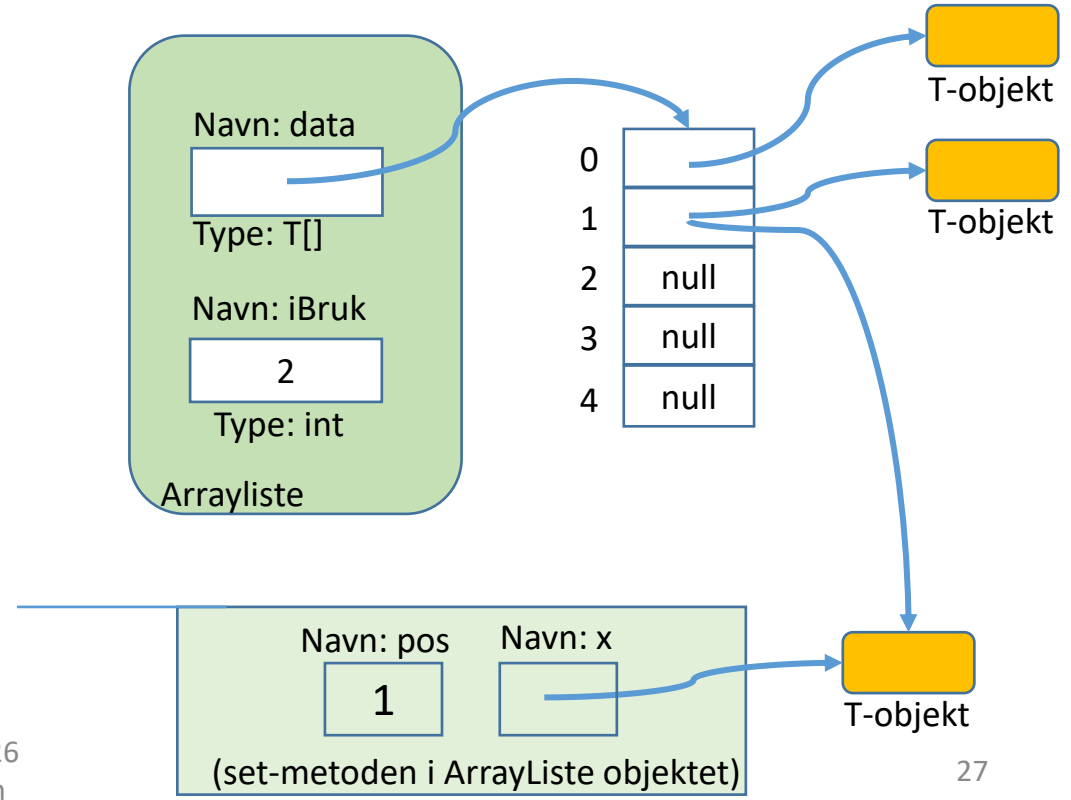
    @Override
    public int size() {
        return iBruk;
    }

    @Override
    public T get(int pos){
        return data[pos];
    }

    @Override
    public void set(int pos, T x){
        data[pos] = x;
    }
}
```

Hva bør sjekkes i get, set?

Se feilhåndtering forrige uke



`Arrayliste`: `add` og `remove`

- Andre metoder og datastruktur, men bruker fremgangsmåten fra klassen `DynamiskStrengArray` forrige uke.

- add
  - Sjekker at det er plass i arrayen, hvis ikke lager vi en ny som er større og flytter over elementene før vi legger inn det nye
- remove
  - Sjekker om lovlig posisjon og velger hvordan vi ellers håndterer det, før vi fjerner elementet

Komplett kode for `Arrayliste` er publisert, lenke fra timeplanen

Varianter av lister

Kø, stabel og prioritetskø

Noen klassiske hovedtyper av lister

Vårt generiske grensesnitt `Liste<T>` har

- operasjoner på bestemte posisjoner (`set, get, remove`)
- `add`

Vi definerte en *liste* som en *ordnet* samling (elementene har en rekkefølge). Rekkefølgen kan utnyttes på ulike måter – noen typiske måter å bruke lister på er

- Kø (henter ut elementet som ble satt inn tidligst)
- Stabel (henter ut elementet som ble lagt inn senest. Tenk tallerkenstabel)
- Prioritetskø (henter ut elementet som haster mest)

Vi skal lage subklasser av `Arrayliste` der `get(0)` gir ulikt resultat avhengig av om subklassen implementerer en kø, stabel eller prioritetskø.

Siden vi alltid tar ut det første elementet må vi ta hensyn til ønsket rekkefølge v ed *innsetting*

Kø («queue»)
First in First Out - FIFO

- En type lister der vi alltid setter inn nye elementer bakerst og henter dem ut fra starten. Eksempelkode (utdrag) for lenket liste.

- Sette inn i kø (bakerst)

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

+ spesialtilfeller og mulige feilsituasjoner

- Hente ut av kø (første element)

```
T ut = null;
if (første != null) {
    ut = start.data;
    første = start.neste;
```

Stabel («stack»)
Last in First Out – LIFO

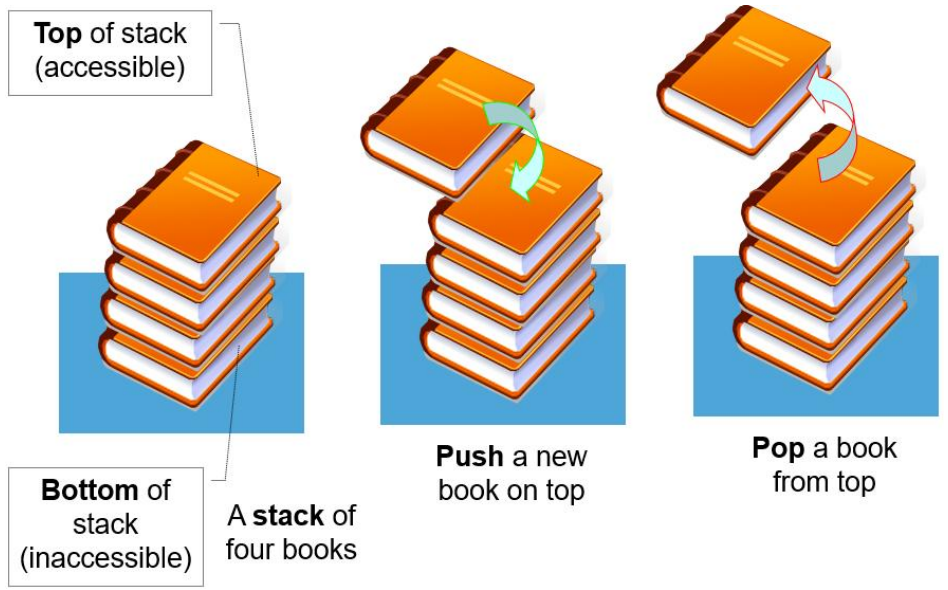
- *Stabler* er lister der vi legger nye elementer på toppen («push-er») og henter fra samme ende («pop-er»).
- Vi jobber hele tiden på toppen = i begynnelsen av listen

- Legge til nytt element («push»)

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

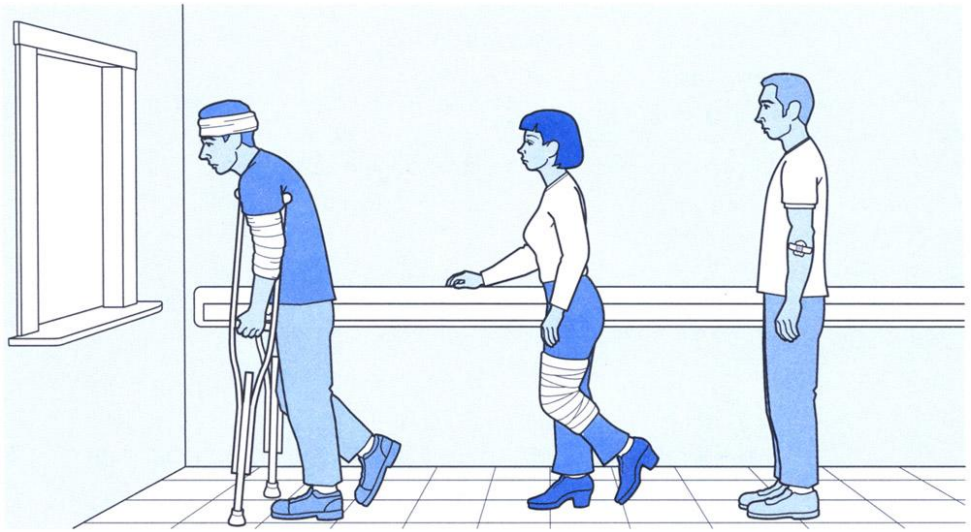
- Ta av et element («POP»)

```
T ut = null;
if (første != null) {
    ut = første.data;
    første = første.neste;
}
```



Prioritetskø («Priority queue»)

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



Implementere en prioritetskø

Invariant for prioritetskøen:
Et element er alltid nærmere starten på listen enn alle elementer med dårligere prioritet

- Vil holde vår prioritetskø sortert ved at *nye elementer* settes på korrekt plass:
  Den første i køen skal være først i listen

- Hvordan vet vi hva som er riktig sortering?

- Vi trenger en standardtest som forteller oss om forholdet for to elementer p og q:
  - Er p < q?
  - Er p = q?
  - Er p > q?

`interface Comparable`

Vi implementerer `Comparable` for å kunne holde listen vår sortert

Eksempel `Comparable`: Resultatliste OL

- Skal rangere landene i vinter-OL etter antall medaljer
- Antall gullmedaljer teller mest
- .. men ved like mange gull, avgjør antall sølvmedaljer
- .. ved like mange gull og sølv, avgjør antall bronsjemedaljer
- .. om de har like mange medaljer i hver valør er to land "like"

- Må kunne sammenligne to og to land
- Trenger en metode som kan kalles på ett objekt med et annet objekt som parameter
- Sammenligningen avhenger av datastrukturen og semantikken i objektenes klasse
  ... dvs denne metoden må skrives på nytt for hver klasse!
- MEN Java interface `Comparable` standardiserer måten å gjøre det på

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);
}
```

- "Vårt" objekt kan sammenlignes med et annet objekt og returnere et heltall
  - `< 0` hvis objektet vi kaller `compareTo` på er "mindre" enn det andre
  - `= 0` hvis objektet vi kaller `compareTo` på er "likt" det andre
  - `> 0` hvis objektet vi kaller `compareTo` på er "større" enn det andre

- Hva som er «minst» og «størst» avgjøres av hver klasse som implementerer grensesnittet

Deltagerland implements Comparable<Deltagerland>

```
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;
        }
    // andre metoder
}
```

Tester sammenligning av deltagerland (2026)

```
class TestMedaljer {
    public static void main(String[] args) {
        Deltagerland danmark = new Deltagerland("Danmark", 0, 1, 0),
                     finland = new Deltagerland("Finland", 0, 1, 5),
                     island = new Deltagerland("Island", 0, 0, 0),
                     norge = new Deltagerland("Norge", 18, 12, 11),
                     sverige = new Deltagerland("Sverige", 8, 6, 4);
        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("Island vs Island: " + island.compareTo(island));
    }
}
```

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

Sorterer vha Comparable

- `Arrays` er en klasse med nyttige array-metoder i Java API
- Metoden `sort` bruker `compareTo` til sortering

```
import java.util.Arrays;
class Resultatliste {
    public static void main(String[] args) {
        Deltagerland[] land = {
            new Deltagerland("Danmark", 0, 0, 0),
            new Deltagerland("Finland", 2, 2, 4),
            new Deltagerland("Island", 0, 0, 0),
            new Deltagerland("Sverige", 8, 5, 5),
            new Deltagerland("Norge 2026", 18, 12, 11),
            new Deltagerland("Norge 2022", 16, 8, 13),
            new Deltagerland("Norge 2018", 14, 14, 11)};
        Arrays.sort(land);
        for (int i = 0; i < land.length; i++)
            System.out.println(land[i].hentNavn());
    }
}
```

```
Danmark
Island
Finland
Sverige
Norge 2018
Norge 2022
Norge 2026
```

!

Implementasjon av prioritetskø

Implementasjon av prioritetskø

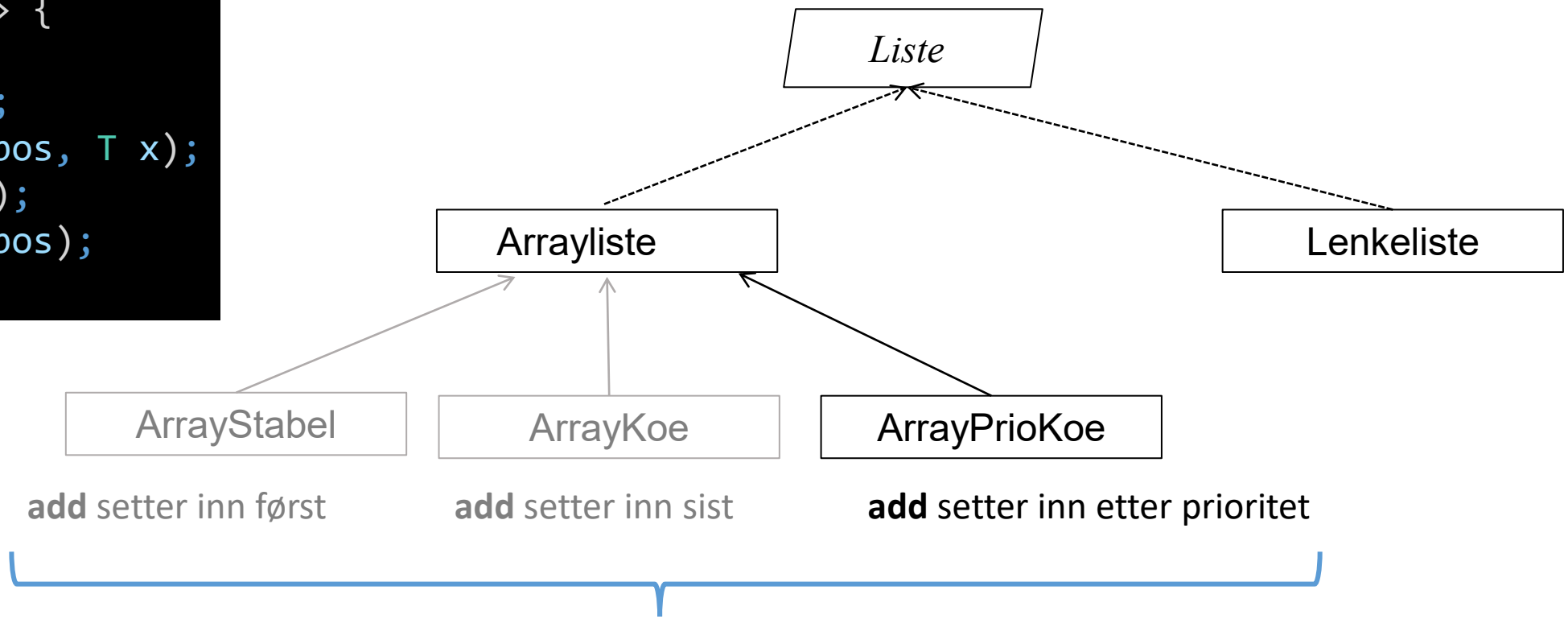
- Vi kan lage `ArrayPrioKoe` som en spesialisering (subklasse) av `Arrayliste`.

- Vi bestemmer oss for en minimal endring av `Arrayliste`:
  - endrer `add` til alltid å sette inn slik at førsteprioritet (lavest verdi) er først i listen
  - `remove(0)` vil ta ut elementet først i køen (som før)
  - `get(pos)` vil la oss se (hente) hvilket element som er nummer `pos` i køen (som før)

Invariant for prioritetskøen:
Et element er alltid
nærmere starten på listen
enn alle elementer med
dårligere prioritet

"Vårt" typehierarki
(ikke det samme som i Java API eller oblig)

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



Klasse som implementerer prioritetskø ved hjelp av `Arrayliste`

- Klassen `ArrayPrioKoe` er en subklasse av `Arrayliste`.

- Typeparameteren `T` må implementere `Comparable` så vi får en sammenligning å sortere etter (og denne sammenligningen må sammenfalle med prioritet)

  (Vi angir alltid en *begrensning* for/ stiller krav til typeparametere med nøkkelordet `extends`)

- Vi redefinerer kun metoden `add` siden innsettingen skal gjøres sortert.
  De øvrige metodene i `Arrayliste` kan være som de er.

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

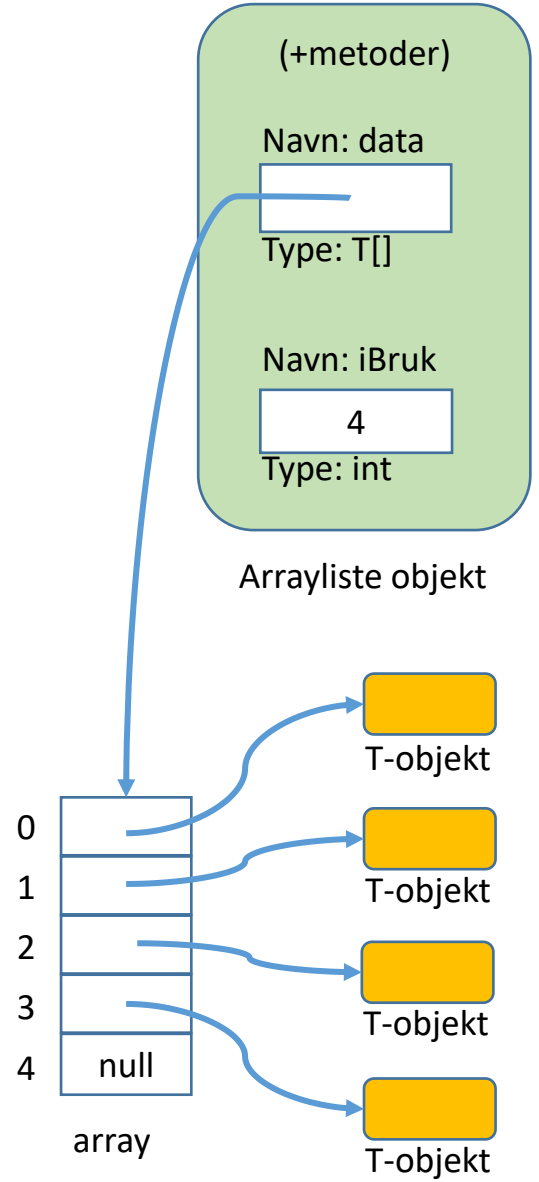
add-metode for prioritetskø

Identifiserer 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.

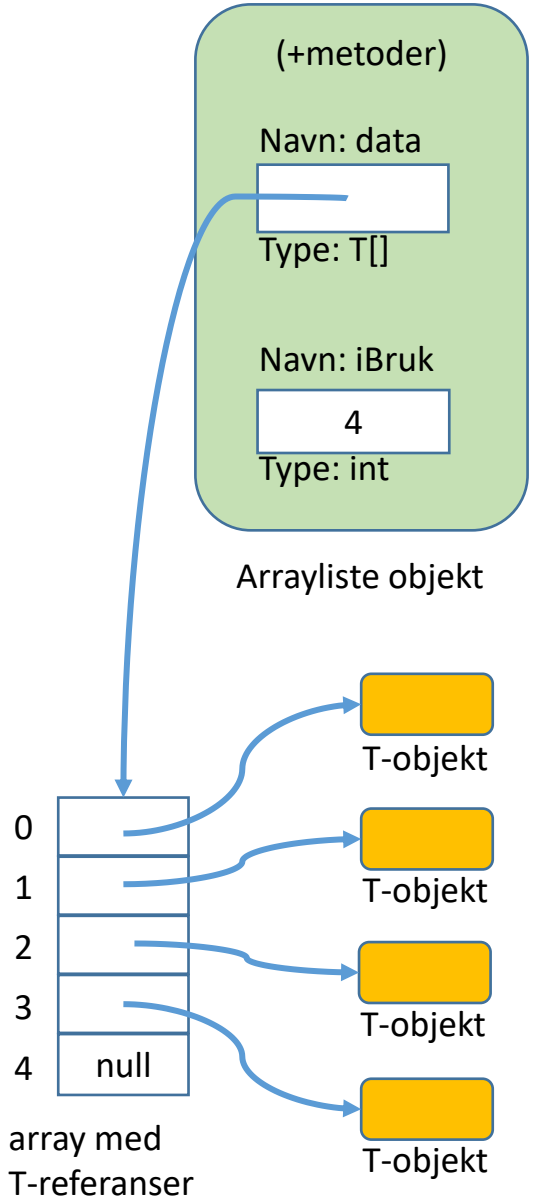


add-metode for prioritetskø (ved tom kø)

```
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 forts. (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 bak.
            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

```
public class TestArrayPrioKoe {
    public static void main(String[] args) {
        Liste<Integer> ap = new ArrayPrioKoe<>();
        ap.add(4);
        ap.add(0);
        ap.add(16);
        ap.add(5);
        ap.add(2);

        ap.remove(0);

        for (int i=0; i<ap.size(); i++) {
            System.out.println(ap.get(i));
        }
    }
}
```

`compareTo` i `Integer`-klassen kalles i metoden `add`

```
2
4
5
16
```

Prioritetskø: Forutsetninger/ valg

- Bruker metoden `compareTo` i interface `Comparable`
  - Hver klasse kan bare ha én `compareTo` metode
  - Hvis denne ikke passer med prioritet må vi finne andre løsninger (ikke IN1010-pensum)

- For vår klasse `ArrayPrioKo` er det en invariant at køen alltid er sortert (som her) – da slipper vi å lete gjennom listen etter høyeste prioritet hver gang

- Merk også at vi sorterer slik at objekter med lavest verdi legges først i køen: Det vil si at lav verdi = høy prioritet

Å iterere gjennom en liste

Java interface `Iterable`

Å gå gjennom en liste

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

```
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 elem: lx) {
    System.out.println(elem);
}
```

- Kalles ofte *enhanced*, i IN1010 *spesialisert* for-løkke. Eller bare *for-each*

=> Hva skal til for å kunne bruke denne på en beholder?

Java interface `Iterable`

- En spesialisert for-løkke itererer gjennom elementene i en beholder som implementerer interface `Iterable`

- interface `Iterable` krever bare at beholderen tilbyr én ekstra metode

- Men hva er en `Iterator<T>` som denne metoden skal returnere?

`Iterator<T>`	`iterator()` Returns an iterator over elements of type T.

(klippet fra Java API, java.lang)

Javas interface `Iterator`

- En iterator kan gi oss elementene i en beholder ett for ett, holde orden på hvor langt vi er kommet i gjennomgangen og sjekke om det er flere elementer igjen

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

interface `Iterator` ligger i Java API (java.util)

- Vi må med andre ord lage en egen `Iterator-klasse` for `Arrayliste` som tilbyr disse metodene for `Arrayliste` beholdere

- `iterator`-metoden i `Arrayliste` kan så returnere et objekt av denne `Iterator`-klassen

Utvider Liste-interfacet med `Iterable`

```
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);
}
```

- Alle `Liste-implementasjoner` må nå være itererbare – dvs tilby metoden `iterator` som returnerer et `Iterator`-objekt for beholderen.
- `Iterator`-objektet må tilby `next` og `hasNext` for beholder-objektet sitt

Implementerer dette i `Arrayliste`-klassen

- Lager en klasse `ListeIterator` som en indre klasse i `ArrayListe`
- Da har den tilgang på alle egenskaper i `ArrayListe` (metoder og instansvariabler)
- Trenger ikke en egen klassevariabel for type – den skal alltid brukes for en beholder-instans der elementene er av type `T` i `ArrayListe`

```
public class Arrayliste<T> implements Liste<T>{
    // instansvariabler

    class ListeIterator implements Iterator<T> {

        @Override
        public T next() { //innhold
        }

        @Override
        public boolean hasNext() { // innhold
        }
    }

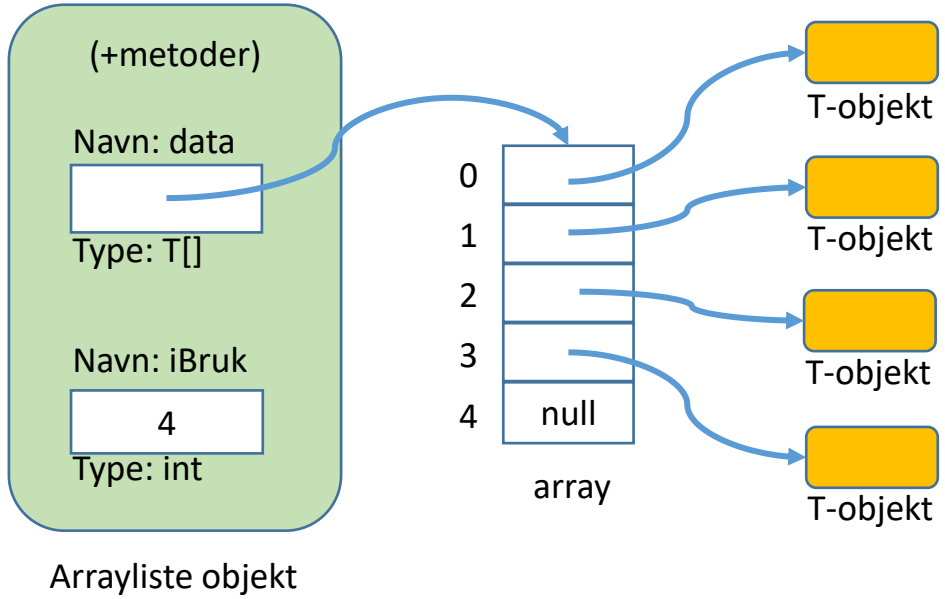
    // Andre metoder
```

Implementerer dette i `Arrayliste`-klassen

- Superklassen `ArrayListe` inneholder datastrukturen for alle subklassene

- Invarianter for alle `ArrayListe` beholderne:
  - `iBruk` angir antall elementer
    - `iBruk>0` betyr at beholderen ikke er tom
  - Elementer er alltid i (riktig) rekkefølge
    - `data[0]` er neste element
    - `data[iBruk-1]` er siste element

- En `Iterator` for `Arrayliste` kan dermed virke for alle subklassene



Klassen `ListeIterator` i `ArrayListe`

```
public class Arrayliste<T> implements Liste<T>{
    // instansvariabler
    class ListeIterator implements Iterator<T> {
        private int pos = 0;
        @Override
        public T next() {
            pos++;
            return get(pos-1);
        }
        @Override
        public boolean hasNext() {
            return pos < size();
        }
    }
    // Andre metoder
```

- Lages et nytt `Iterator`-objekt før hver gjennomgang av elementene i en beholder
- Iteratoren trenger en posisjonspeker som holder rede på hvor langt vi er kommet (instansvariabel `pos`)
- Bruker metodene `get` og `size` i `ArrayListe` (kunne også aksessert instansvariablene direkte)

Metoden `iterator()` i `Arrayliste`

```java
@Override
public Iterator<T> iterator() {
    return new ListeIterator();
}
```

- `ListeIterator`-objektet som returneres har tilgang til egenskapene i det `Arrayliste`-objektet metoden ble kalt på

- Java kan nå tilby *for-each* løkker på `Arrayliste`-beholdere, fordi vi tilbyr et Iterator-objekt med `next`- og `hasNext`-metoder.

- (Programmerere som bruker `Arrayliste`-beholderen vår kan også be om et iterator-objekt for å kalle `next` og `hasNext` på sin beholder - må importere `java.util.Iterator`)

Itererbar Arrayliste – endret:

- `Liste<T> extends Iterable<T>`
  - Utvidet med metoden `iterator`
- Arrayliste
  - implementerer metoden `iterator`
  - .. som oppretter `Iterator`-objekt for `Arrayliste`
  - .. fra den indre klassen `ListeIterator` som tilbyr `hasNext`- og `next`-metoder
- TestArrayliste kan nå
  - bruke for-each (Java oppretter og bruker Iterator-objekt bak kulissene) - eller
  - lage et eget Iterator-objekt for å hente ut elementer i rekkefølge

```java
import java.util.Iterator;
public class Arrayliste<T> implements Liste<T>{
    @SuppressWarnings("unchecked")
    private T[] data = (T[]) new Object[10];
    private int iBruk = 0;

    class ListeIterator implements Iterator<T> {
        private int pos = 0;
        @Override
        public T next() {
            pos++;
            return data[pos-1];
        }
        @Override
        public boolean hasNext() {
            return pos < iBruk;
        }
    }

    @Override
    public Iterator<T> iterator() {
        return new ListeIterator();
    }

    // Andre metoder
```

Tester Arrayliste med iterator

```java
import java.util.Iterator;

public class TestArrayliste<T> {
    public static void main(String[] args) {
        Liste<String> lx = new Arrayliste<>();
        // Sett inn 13 elementer:
        for (int i = 0; i < 13; i++) {
            lx.add("A" + i);
        }
        // test av andre metodene utelatt her
        // (komplett kode på semestersiden)

        for (String elem: lx) {
            System.out.println(elem);
        }

        System.out.println("Tester eksplisitte kall på iterator:");
        Iterator<String> iter = lx.iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }
    }
}
```

```
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10*
A11
A12
Tester eksplisitte kall på iterator:
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10*
A11
A12
PS Microsoft.PowerShell.Core\FileSystem::\
```

Hva med beholderne som arver Arrayliste?

```java
public class TestArrayPrioKoe {
    public static void main(String[] args) {
        Liste<Integer> ap = new ArrayPrioKoe<>();
        ap.add(4);
        ap.add(0);
        ap.add(16);
        ap.add(5);
        ap.add(2);

        ap.remove(0);

        System.out.println("Tester iterable");
        for (Integer x: ap) {
            System.out.println(x);
        }
    }
}
```

```
>java TestArrayPrioKoe
Tester iterable
2
4
5
16
```

Oppsummering

- Flere mulige (og gode) datastrukturer for lenkelister

- Har beskrevet og delvis implementert arraylister med ulik semantikk
  - Kø (tar ut foran, setter inn bakerst)
  - Stabel (tar ut foran, setter inn foran)
  - Prioritetsliste (tar ut foran, setter inn etter prioritet)

- Mer Java
  - Innpakking ("boxing")
  - Å sammenligne objekter (Comparable)
  - Å gå gjennom alle elementer i en beholder (med Iterator)

- Neste uke: Designmønstere v/ Eyvind

  Prøveeksamen legges ut
  Vi flytter til Simula i O-JD!

