Beholdere og generiske klasser I

Uke 6

28. februar 2023

Beholdere og generiske klasser - I

- Hvorfor og hvordan velge og bruke beholdere?
- Klassehierarkier for beholdere
- Et egendefinert klassehierarki for beholdere med **interface Liste**

- Nye Java mekanismer
  - Klasseparametere (typeparametere) og generiske klasser (generics)
  - Indre klasser
  - Egne Exceptions: Deklarasjon, opprettelse og behandling

Dagens forelesning dekker det meste av oblig 3. Neste uke gir grunnlag for Del F og flere eksempler på hvordan vi kan representere og manipulere lenkelister.

Hva legges på semestersiden?

- Hele koden for **`class Arrayliste`**
- Testprogrammet
- Lysarkene
- (ikke mer av **`class Lenkeliste`** enn det som er gitt på lysarkene)
- + pensum, Trix-oppgaver, opptak,..

Hvorfor trenger vi beholdere?

- Fra ordlisten for IN1000/ IN1010 (lenke fra semestersiden)

Container	Beholder (f.eks. liste, ordbok eller mengde) for en samling elementer eller objekter

- Når vi skal ta vare på og arbeide med "mange" verdier (referanser til objekter)

- Hvordan ville det vært å programmere uten beholdere? Uten lister, arrayer, ArrayList, HashMap...
- Om vi f eks skal lese inn en rekke navn fra terminal eller fil
  - Lage en variabel for hver eneste verdi vi trenger??
  - Lage et nytt objekt å lagre hver verdi i? Hvor lagrer vi referanser til disse objektene?
- Med array:
  - Kan lage et "uendelig" antall like plasser – men **må vite ved kjøring, før bruk,** hvor mange

Hvorfor er beholdere ("collections") pensum i IN1010?

- Det er nyttige verktøy for svært mange programmer (det har dere allerede sett i IN1000/ IN1900).
- For å velge optimale verktøy bør dere kjenne til hvordan de er bygget opp og fungerer.
- Dere kan få behov for å skrive lignende selv.

=> Dette er veldig gode eksempler på og trening i objektorientert programmering.

Beholdere har ulike funksjonelle og ikke-funksjonelle egenskaper

- Grensesnittet (inkl semantikken i metodene) definerer de funksjonelle egenskapene
  - "Hva kan jeg gjøre med denne beholderen, hvor lett kan jeg få gjort det jeg trenger?"

- De ikke-funksjonelle egenskapene (typisk effektivitet) avhenger av implementasjonen
  - Minnebruk, prosessortid, evt andre ressurser

Java API, denne forelesningen og oblig3 definerer *hvert sitt klassehierarki* for beholdere.

Valg av beholder

- I IN1010 (og mange applikasjoner) betyr effektivitet lite for valg av løsning
  ... men dere lærer noen viktige begreper og mekanismer for å forstå forskjeller og gjøre valg – og for å implementere deres egne

- (i IN2010 lærer man om hvordan man kan beregne og optimalisere ressursbruk for typiske operasjoner)

- Vi skal se på
  - grensesnittet til noen klassiske *typer* av beholdere
  - hvordan typer av beholdere kan struktureres i et klassehierarki ved hjelp av arv og interface
  - to ulike løsninger for implementasjon av beholdere med variabelt antall elementer

Grensesnittet til en beholder

- Syntaks (metodenavn, parametere, returtyper)

- Semantikk – hvordan virker metodene

Eksempel: Hvilket element tas ut om du ikke spesifiserer vha. parameter

- Ta ut elementet som har ligget lengst (First in First Out, *kø*)
- Ta ut elementet som ble lagt inn sist (Last in First Out, *stabel*/ stack)
- Ta ut elementet med lavest/ høyest verdi for en egenskap (*prioritetskø*)

Java Collections Framework

Verktøy for lagring og organisering av objekter

Java dokumentasjonen: *A collections framework is a unified architecture for representing and manipulating collections*

Hierarki av interface- og klassetyper for beholdere.

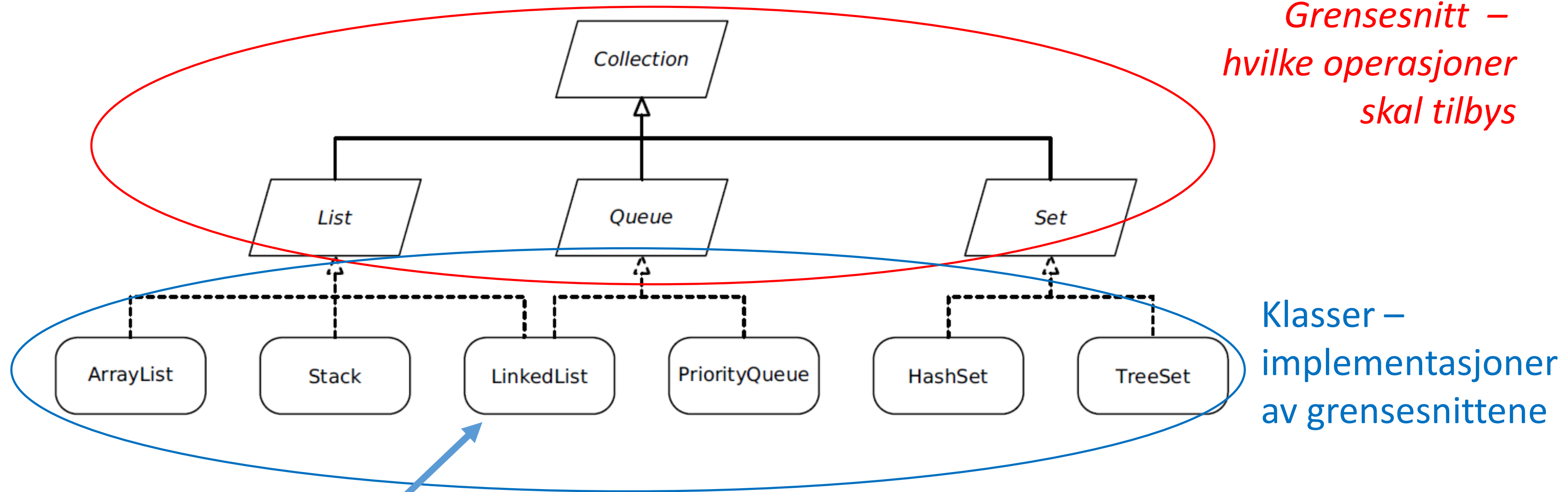
Collection Interface er et felles grensesnitt for lister (med rekkefølge) og mengder (uten rekkefølge)

Map er grensesnitt for organisering av nøkkel->verdi par.

- Mange klasser som implementerer et eller flere grensesnitt
- Mange grensesnitt som er implementert av en eller flere klasser

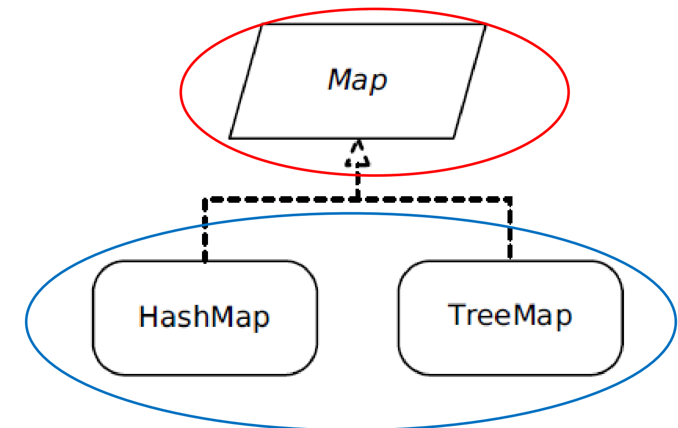
Java Collections Framework (utdrag! fra Big Java)

Grensesnitt – hvilke operasjoner skal tilbys



Klasser – implementasjoner av grensesnittene

Klassen LinkedList implementerer (her) 2 grensesnitt

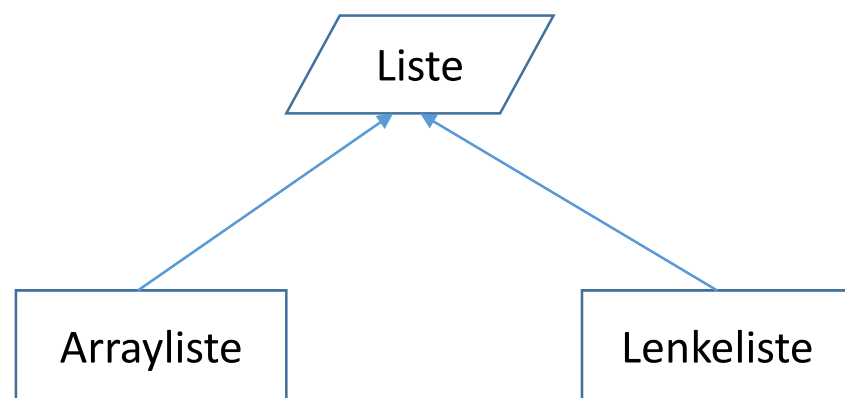


Et eget **`interface Liste`**

.. og to eksempler på implementasjon

Et klassehierarki for beholdere

Forelesnings-eksempel – ikke samme hierarki som i oblig 3 eller Java API



Grensesnitt for vårt interface Liste

- Deklarerer et interface som kan brukes som felles type for ulike implementasjoner av beholdere
- interfacet Liste skal tilby operasjoner som forholder seg til posisjon i listen:

```
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
}
```

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

Hvordan lage en generisk beholder?

Forslag 1

- **Object** som type for elementene – da kan objekter av alle klasser legges inn

- Dette virker – men krever typekonvertering når vi henter ut elementer som skal brukes videre

- Bruker av klassen må selv passe på at listen kun inneholder riktige typer
  => <u>usikker løsning</u>

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

Bruk av beholder som implementerer Liste:

```
String element = (String) minListe.get(10);
```

Forslag 2: Klasseparameter

- Vi har brukt parametere for å få en metode til å bruke en ny verdi (av samme type) for hver gang den blir kalt – i stedet for å skrive en egen metode for hver tenkelige verdi (ikke gjennomførbart!)

- Klasser i Java kan ha parametere som angir en *type* (klasse) som skal brukes (inne) i en bestemt instans av klassen. Dette kaller vi *generiske klasser* med *klasseparametere*

- Brukes f.eks. i ArrayList:

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

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

Deklarasjon og bruk av generisk klasse

- En parameter i en klassedeklarasjon (formell parameter) angir at klassen kan bruke referanser til ulike klasser eller interface
- Kalles *klasseparameter* eller *typeparameter*

- Når vi lager en instans av klassen bestemmer vi hvilken type denne instansen (objektet) skal jobbe med
- Kalles *aktuell parameter/ argument*

- Veldig nyttig for beholdere!

```
class Beholder<E> {
    E element1;
    E element2;

    public void settInn(E ny1, E ny2) {
        element1 = ny1;
        element2 = ny2;
    }

    public E taUt1() {
        return element1;
    }

    public static void main(String[] args) {
        Beholder<String> b = new Beholder<>();
        b.settInn("A", "B");
        System.out.println(b.taUt1());
    }
}
```

Om klasseparametere

- Interface kan ha (og være) klasseparameter på samme måte som klasse
- Klasseparameter kalles også *typeparameter*
  (kan angi klasse eller interface, for en klasse eller et interface)

- Navnekonvensjon for typeparametere (fra Java doc):
  - 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*

Et generisk Liste-interface (grensesnitt)

- Grensesnittet kan ha en typeparameter som representerer typen til 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);
}
```

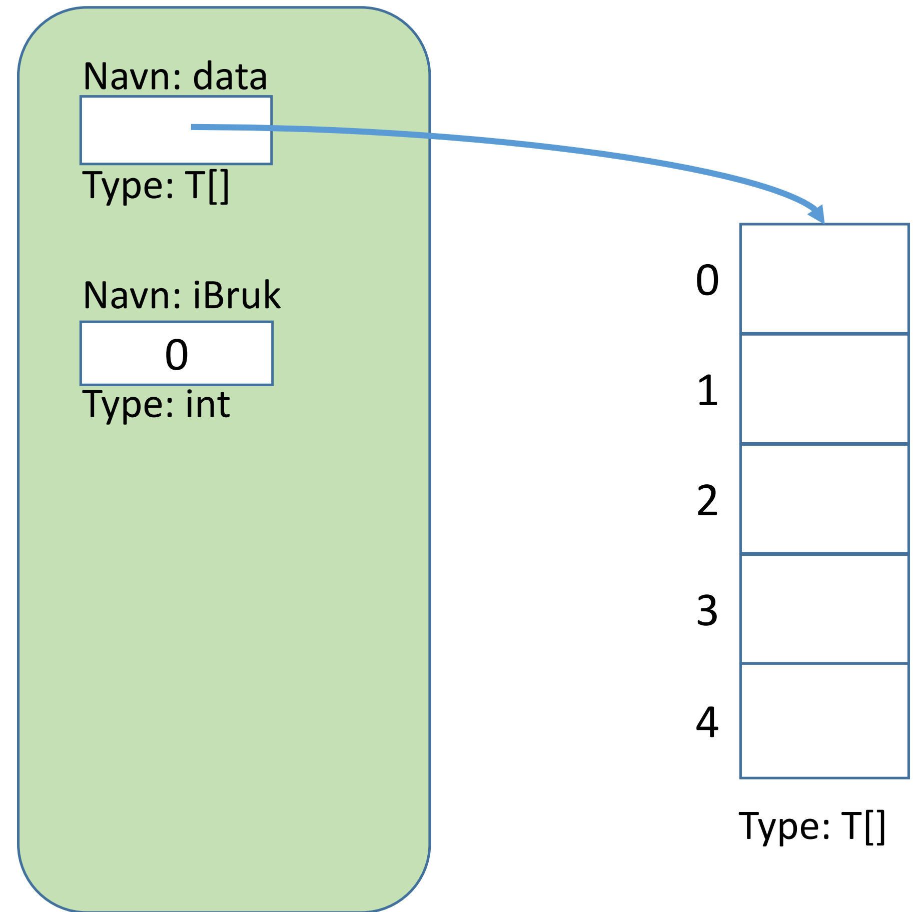
Fortsatt ikke samme klassehierarki og klasser som i oblig 3 eller Java API

Liste-interface implementert med array

Full implementasjon av Arrayliste. Koden med testprogram ligger på uke6-siden.

Bruk av array for å lagre elementene

- Skriver en egen klasse **Arrayliste**
- Implementerer interface **Liste**
- Lagrer elementene i en array
- Velger en lengde på arrayet – her 5



Datastruktur – 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];
```

tilbake til Object[] som type for arrayen

men 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

```
ressWarnings("unchecked")
te T[] data = (T[]) new Object[5];
```

Datastruktur for Arrayliste klasse

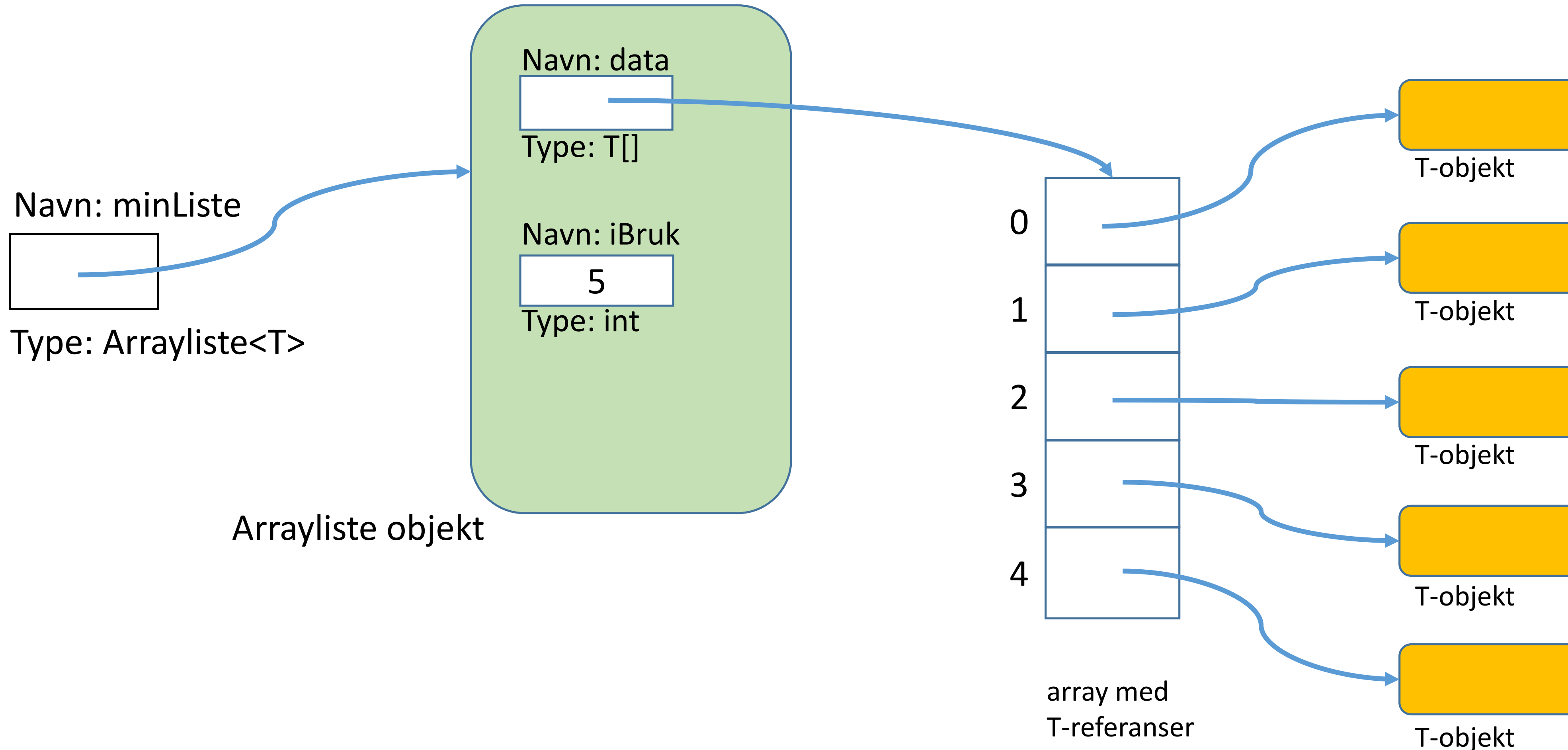
```
public class Arrayliste<T> implements Liste<T> {
    protected T[] data = (T[]) new Object[5];
    protected int iBruk = 0;
```

- En array til å lagre elementene i
- Vi bruker bare så mange plasser i arrayen vi trenger
  => iBruk forteller hvor mange plasser som er i bruk
  OG hva som er neste ledige plass

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

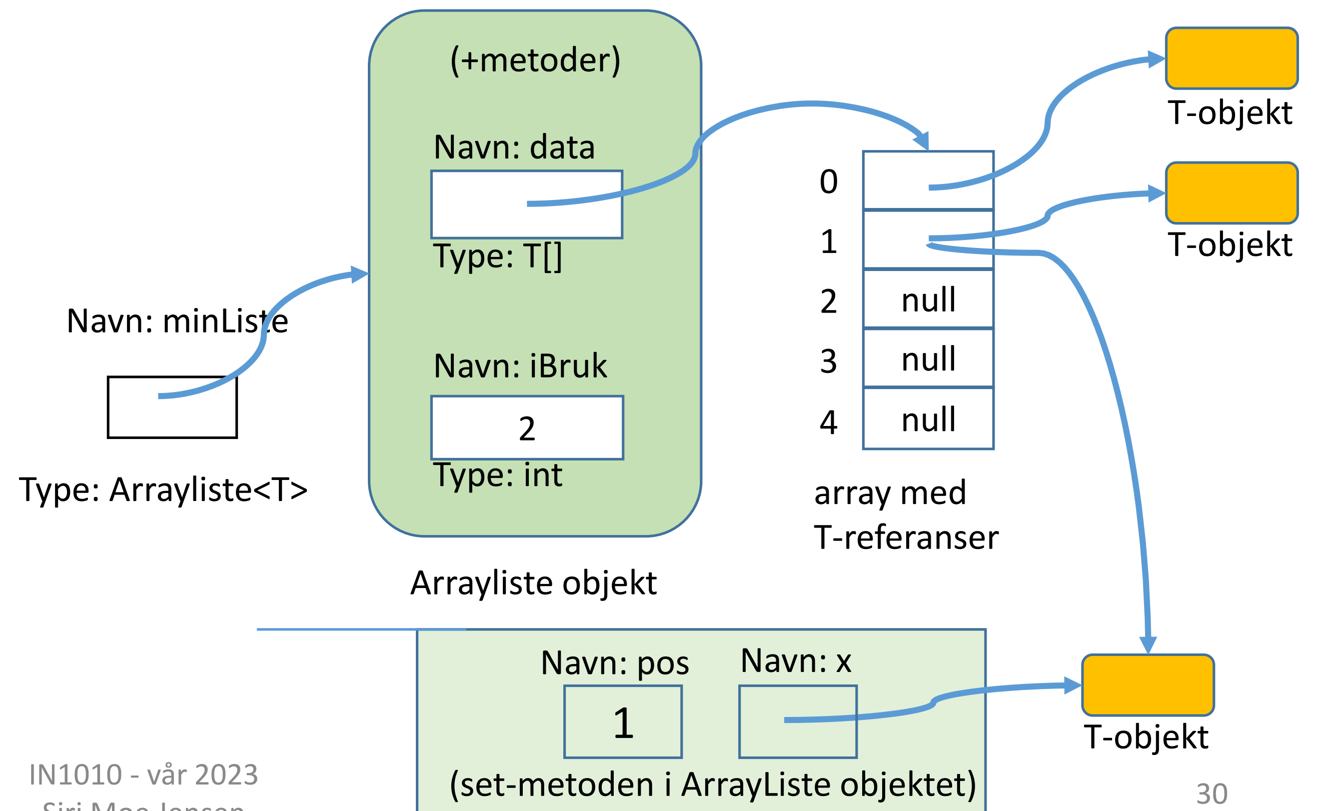


Klassen Arrayliste: size, set, get

```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;
    }
}
```

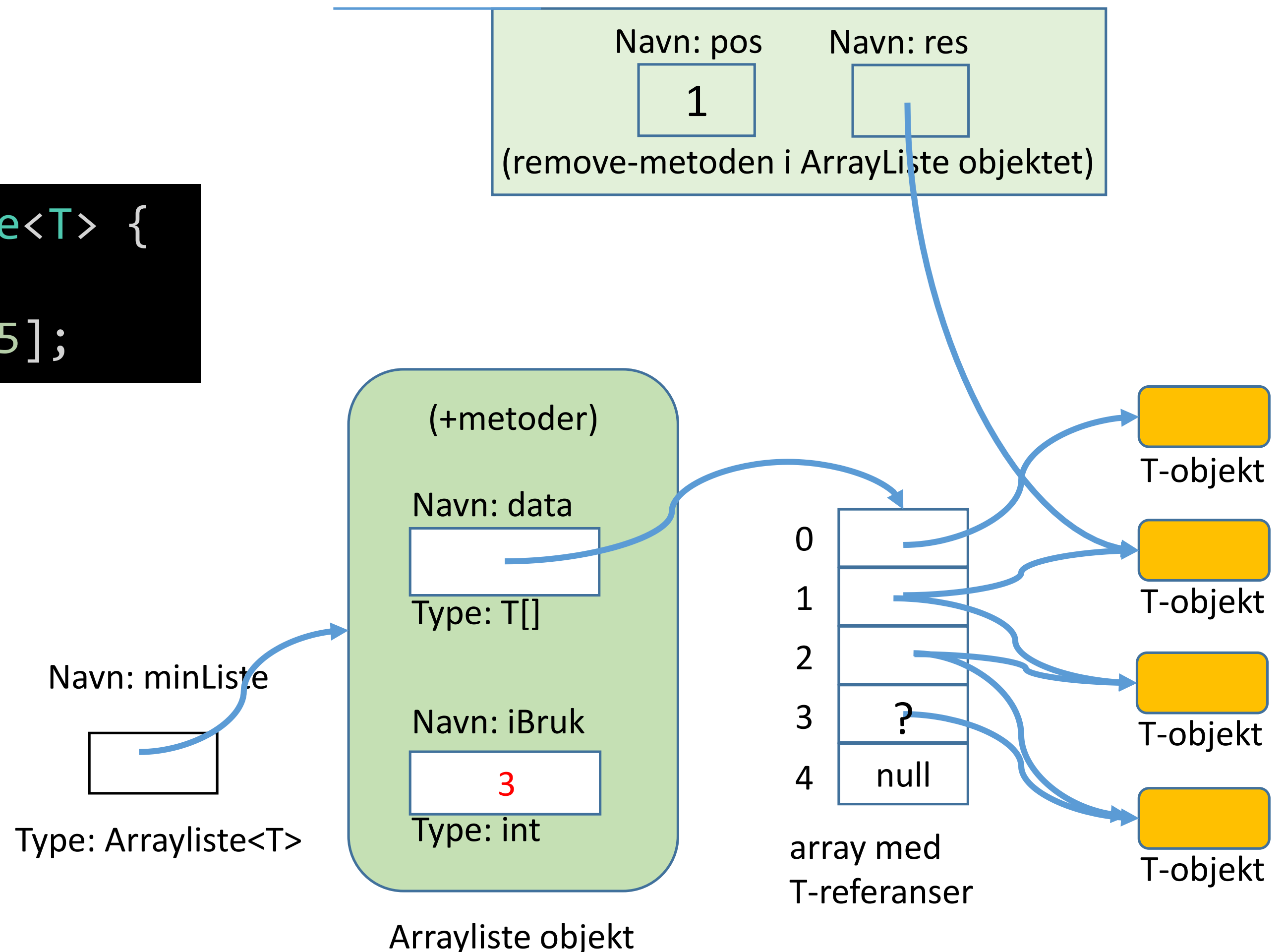
Kan noe skape problemer for get?

Kommer tilbake til mulige feilsituasjoner

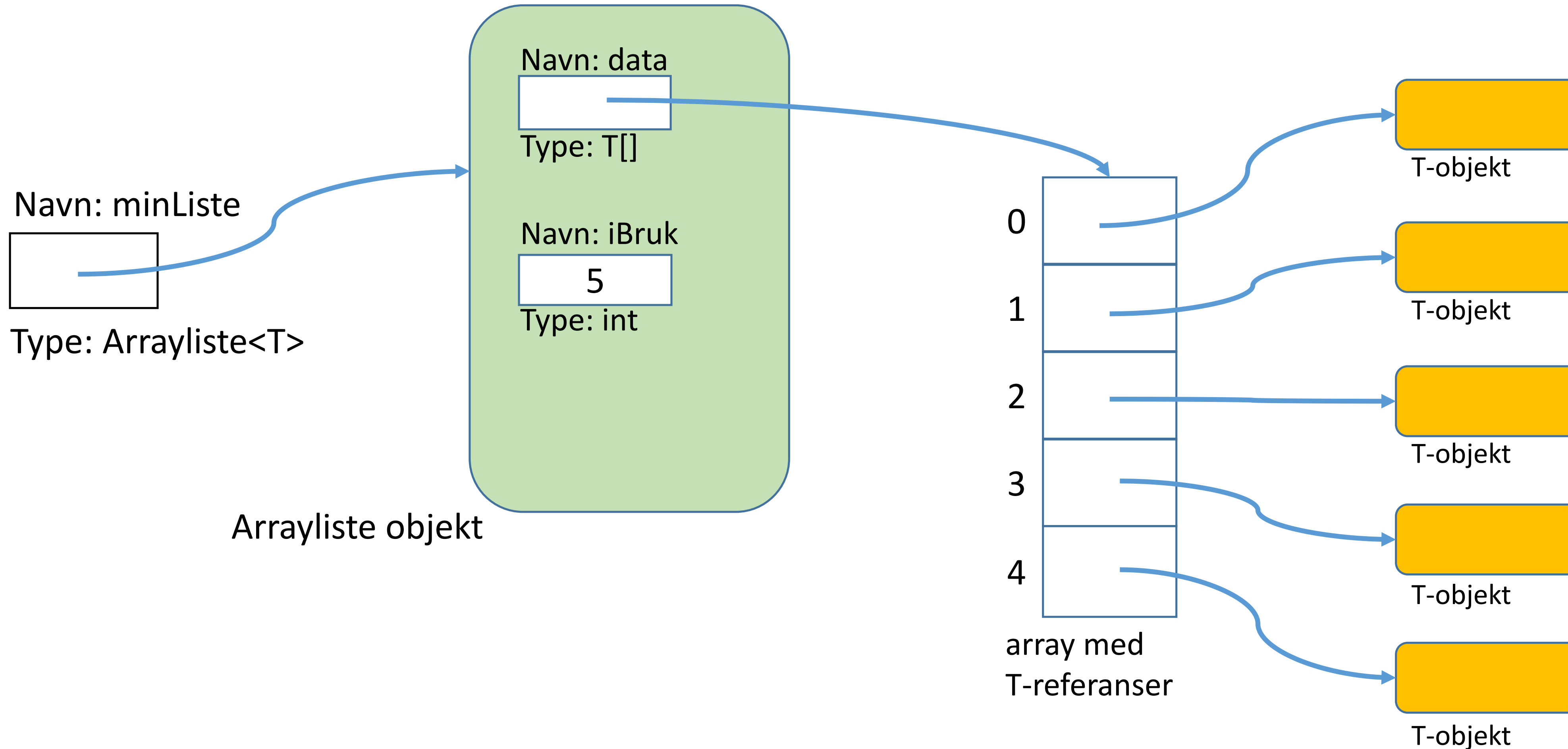


Klassen Arrayliste II: remove

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

        // andre metoder

    @Override
    public T remove(int pos) {
        T res = data[pos];
        for (int i=pos+1; i<iBruk;i++) {
            data[i-1]=data[i];
        }
        iBruk--;
        return res;
    }
}
```



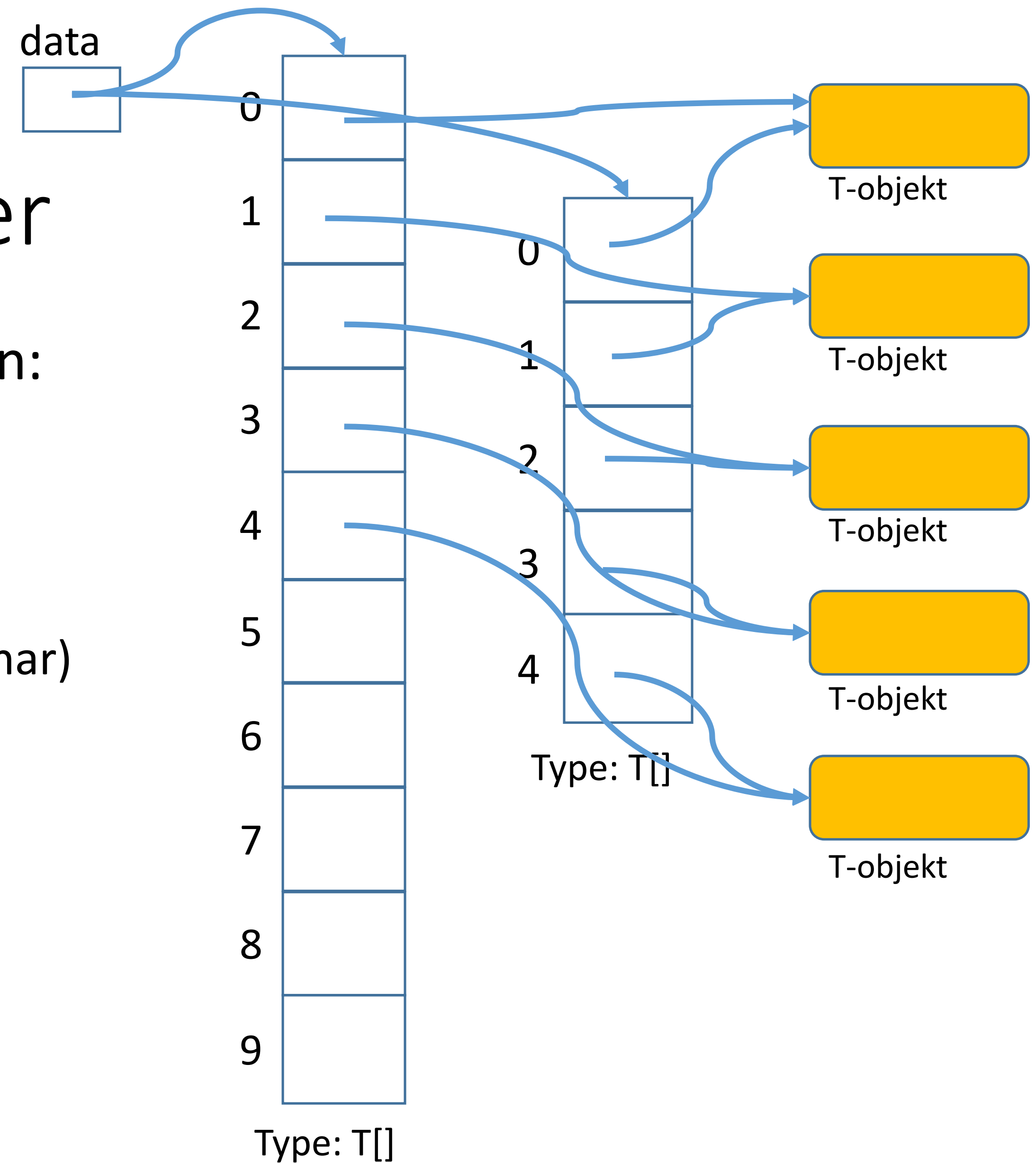
Arrayliste objekt med full array



Klassen Arrayliste:
Lage plass til flere elementer

- Spesialtilfelle ved tillegg nytt element i listen:
  - arrayen som holder dataene kan være full!

- Må da allokere mer plass =>
  - oppretter ny array med flere plasser (2* den vi har)
  - flytter eksisterende elementer over
  - legger til det nye på første ledige plass



Klassen Arrayliste III (add)

```
@Override
public void add(T x){
    if (iBruk == data.length) {
        @SuppressWarnings("unchecked")
        T[] ny = (T[]) new Object[2*iBruk];
        for (int i=0; i<data.length;i++) {
            ny[i] = data[i];
        }
        data = ny;
    }
    data[iBruk] = x;
    iBruk++;
}
```

Invariant (se også forrige ukes forelesning)

- Invarianter er nyttige når vi skal sikre oss mot feil i programmene våre: Hjelpemiddel for mer systematisk analyse

- En invariant er en påstand som alltid gjelder på et bestemt sted (før/ etter en gitt programsetning/ blokk) under utføring av et program
- En invariant sier noe (men ikke nødvendigvis alt) om programmets *tilstand* på det aktuelle punktet.

- Invarianter kan for eksempel si noe om hvilke (kombinasjoner av) instansvariabel-verdier som er lovlige mellom metodekall på et objekt

Invarianter for Arrayliste

Gjelder alltid mellom metodekall

- iBruk angir hvor mange elementer beholderen har
- arrayen **data** er alltid fyllt fra indeks 0 til iBruk-1
- **iBruk == data.length** betyr at arrayen er full!
    - ellers er iBruk indeks for neste ledige plass

- Nyttig å formulere mens man designer datastrukturen for ArrayListe?
- Nyttig å minne seg på under implementasjon av ArrayListe?

Testprogram Arrayliste

(kun eksempel – tester ikke alle situasjoner)

TestArrayliste.java

```
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);
        }
        // sjekk antall
        System.out.println("Listen har " + lx.size() + " elementer");
        // Marker element nr 10:
        lx.set(10, lx.get(10) + "*");
        // Fjern første element
        lx.remove(0);
        System.out.println("Fjernet element 0");
        // Skriv ut listen
        for (int i = 0; i < lx.size(); i++) {
            System.out.println("Element " + i + ": " + lx.get(i));
        }
        // Lag en feil
        lx.remove(999); }
}
```

Kjøring av test

```
Siri>java TestArraylisteFeil
Listen har 13 elementer
Fjernet element 0
Element 0: A1
Element 1: A2
Element 2: A3
Element 3: A4
Element 4: A5
Element 5: A6
Element 6: A7
Element 7: A8
Element 8: A9
Element 9: A10*
Element 10: A11
Element 11: A12
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
 Index 999 out of bounds for length 20
        at ArraylisteFeil.remove(ArraylisteFeil.java:39)
        at TestArraylisteFeil.main(TestArraylisteFeil.java:19)
```

- Det meste går bra, men:

=> gal parameter til remove gir en lite brukervennlig feilmelding.

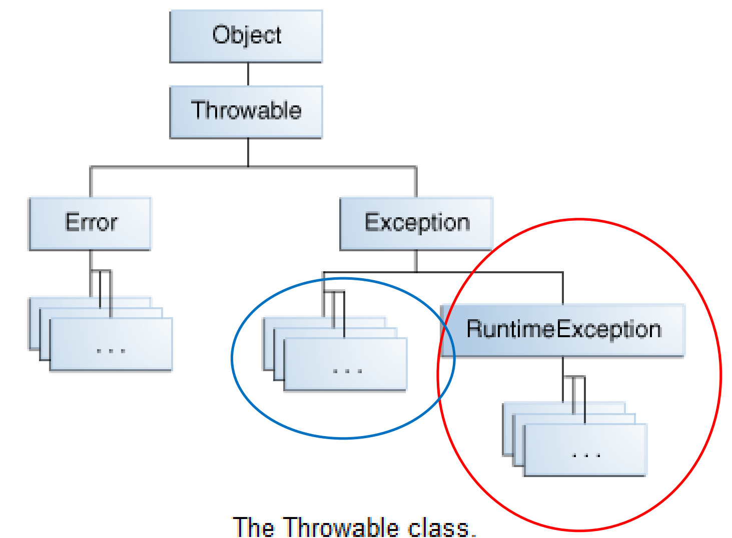
- (Det gjelder også get og set.)

Feilhåndtering

- Hva kan gå galt?
  - Feil i Java => Error, trenger ikke tenke på/ håndtere
  - Feil (bugs) i din kode eller ulovlig input

- checked (eks: prøver å åpne ikke-eksisterende fil)
  - Java *krever* håndtering
  - kan fanges og håndteres lokalt eller **throw** videre bakover
  - ved **throw** må det angis i metodesignaturen (... **throws** ...)

- unchecked (eks: ArrayIndexOutOfBoundsException)
  - du velger om du vil teste og evt. håndtering
  - kan, men trenger ikke angi i metodesignaturen om du kaster



The Throwable class.

Feilhåndtering med Exceptions

- Der feilen oppdages:
  - Håndter unntaket der og da (se under), eller
  - kast passende Exception (**throw**) til kallsted

- Teste og fange (**try** – **catch**) Exceptions
  - Kan velge hvordan de håndteres, f eks:
    Be om ny verdi fra bruker, avslutte en operasjon eller avslutte programmet
    Gjerne skrive ut/ logge en feilmelding med detaljer
  - Om du *ikke* fanger en Exception som oppstår, håndterer Java den "på sin måte":
    Avslutter programmet med en feilmelding

Egne Exceptions (kan også bruke innebygde)

- Feilmeldinger bør være en subklasse av passende Exception
- Her: RuntimeException eller en subklasse(se Exception klasse-hierarki med forklaringer i Big Java eller Java API).
- Konstruktøren tar parametere med nyttig informasjon om feilen (her: hvilken indeks ble brukt, og hvilke er lovlige)

```
class UlovligListeIndeks extends RuntimeException {
    public UlovligListeIndeks(int pos, int max) {
        super("Listeindeks " + pos + " ikke i intervallet 0-" + max);
    }
}
```

Oppdage at noe er feil

- Vi tar vare på relevant informasjon der feil kan oppstå (for eksempel i metoden remove)
- ... og sender den med til Exception-objektet vi oppretter
- ... som vi så "kaster" tilbake til kallstedet med **throw**

```
    @Override
    public T remove(int pos) {
        if (pos<0 || pos>=iBruk) {
            throw new UlovligListeIndeks(pos, iBruk-1); }
        T res = data[pos];
        for (int i = pos + 1; i < iBruk; i++) {
            data[i - 1] = data[i]; }
        iBruk--;
        return res; }
```

Håndtere feil som oppstår i en metode

- Når vi bruker metoder som kan kaste unntak (som remove) skriver vi en try - catch blokk for å håndtere dem

```
// Lag en feil - og håndter den
try {
    lx.remove(999);
} catch (UlovligListeIndeks u) {
    System.out.println("Feil: " + u.getMessage()); }
```

- Hvordan vet vi om andres metoder kaster unntak?
  - Unchecked: Metode-signatur, dokumentasjon, eller "for sikkerhets skyld"
  - Checked: Alltid i metode-signaturen

Arrayliste med eksempel på tilpasset feilhåndtering

(merk at vi fortsatt ikke dekker alle feilsituasjoner som kan oppstå med gode feilmeldinger)

```
public class Arrayliste<T> implements Liste<T> {
    @SuppressWarnings("unchecked")
    protected T[] data = (T[]) new Object[10];
    protected int iBruk = 0;

    @Override
    public void set(int pos, T x)
                throws UlovligListeIndeks {
        if (pos<0 || pos>=iBruk) {
            throw new UlovligListeIndeks(pos, iBruk-1); }
        data[pos] = x; }

    @Override
    public T remove(int pos)
                throws UlovligListeIndeks {
        if (pos<0 || pos>=iBruk) {
            throw new UlovligListeIndeks(pos, iBruk-1); }
        T res = data[pos];
        for (int i = pos + 1; i < iBruk; i++) {
            data[i - 1] = data[i]; }
        iBruk--;
        return res; }
}
```

Testprogram

```
>java TestArrayliste
Listen har 13 elementer
Fjernet A0
Element 0: A1
Element 1: A2
Element 2: A3
Element 3: A4
Element 4: A5
Element 5: A6
Element 6: A7
Element 7: A8
Element 8: A9
Element 9: A10*
Element 10: A11
Element 11: A12
Feil: Listeindeks 999 ikke i intervallet 0-11
Fortsetter etter catch-blokken
```

```java
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);
        }
        // sjekk antall
        System.out.println("Listen har " + lx.size() + " elementer");
        // Marker element nr 10:
        lx.set(10, lx.get(10) + "*");
        // Fjern første element
        lx.remove(0);
        System.out.println("Fjernet element 0");
        // Skriv ut listen
        for (int i = 0; i < lx.size(); i++) {
            System.out.println("Element " + i + ": " + lx.get(i));
        }
        // Lag en feil - og håndter den
        try {
            lx.remove(999);
        } catch (UlovligListeIndeks u) {
            System.out.println("Feil: " + u.getMessage()); }
        System.out.println("Fortsetter etter catch-blokken");
    }
}
```

Alternativ implementasjon av Liste – samme grensesnitt

Noen tips for implementasjon med lenkeliste

Kan vi implementere Liste på en annen måte?

- I forrige eksempel implementerte vi interface Liste ved hjelp av klassen Arrayliste
- Arrayliste bruker en array som datastruktur for objektene – krevde håndtering av fullt array

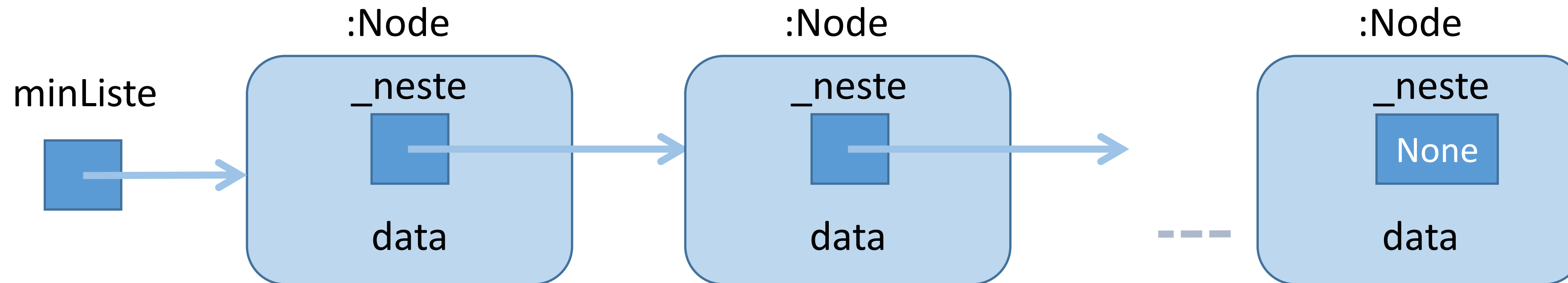
- Kan vi lage en beholder som lagrer objekter på en mer dynamisk måte – der vi alltid kan ta inn *ett til* uten å "bygge om" datastrukturen vår?
- Det vi skal lagre (objektene) ligger utenfor beholder-klassen, det vi trenger er en datastruktur der det alltid er *en ledig referanse* til det nye elementet

$\Rightarrow$ for hvert element, oppretter vi et hjelpe-objekt (en node)

$\Rightarrow$ som skal referere til det nye elementet –

$\Rightarrow$ OG kan referere til et nytt hjelpeobjekt (node)

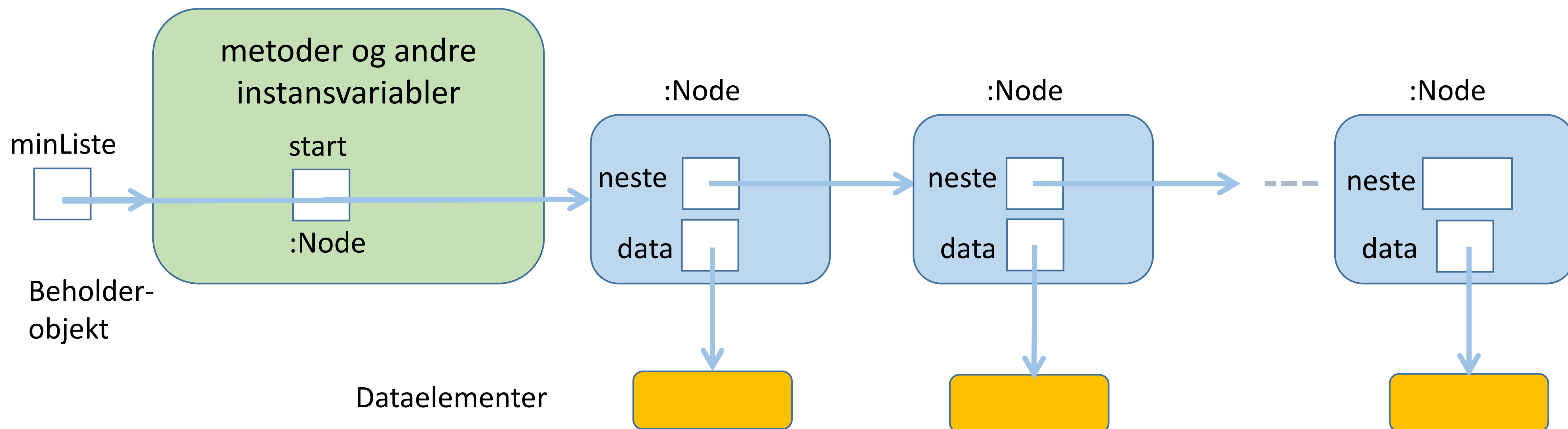
Lenkeliste (NB: figur fra IN1000, Python)



- Poenget med en lenkeliste er at for hvert nye objekt vi lager – så lager vi samtidig en referansevariabel som kan referere til enda et nytt objekt
- dvs hvert objekt må kunne referere til et annet objekt
- dermed får vi en lenket liste av objekter – og trenger bare ha én referanse, til det første objektet

Beholder basert på lenkeliste-struktur

- Vi lager en beholder-klasse som "pakker inn" datastrukturen (nodene med neste-referanser) og metodene i grensesnittet
- Den som bruker beholderen kan bruke metodene i grensesnittet til å legge inn og ta ut data uten å kjenne til strukturen (lenkelisten) og implementasjon av metodene



Datastruktur inne i et Lenkeliste-objekt
(erstatter arrayen vi brukte i Arrayliste)

- Vi trenger en Node-klasse
- Hvert Node-objekt trenger
  - en referanse til et data-objekt (element i beholderen)
  - en referanse til et nytt Node-objekt

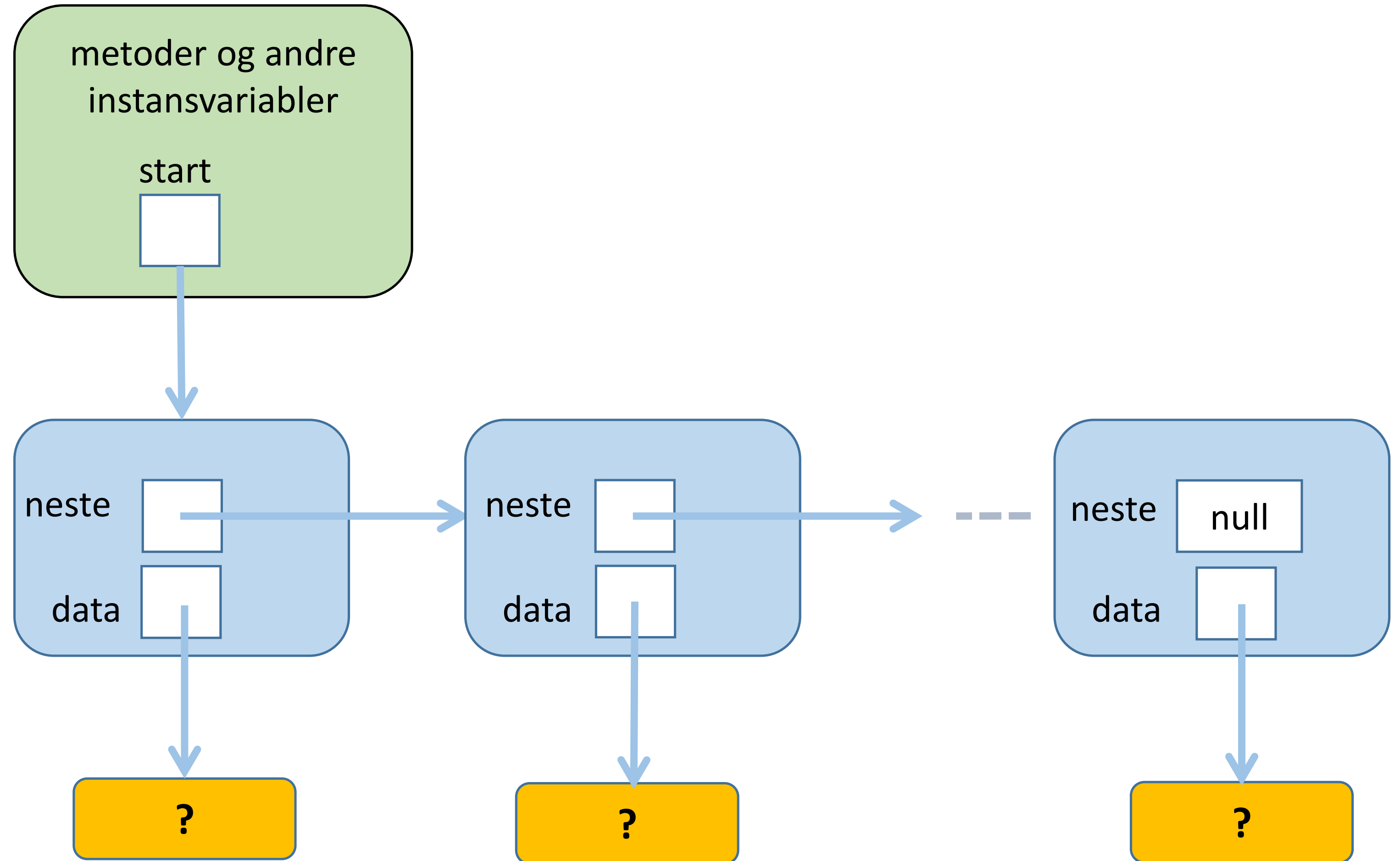
- Beholderen trenger (minst)
  - en referanse til det første Node-objektet

Datastruktur *inne i* et Lenkeliste-objekt (erstatter arrayen vi brukte i Arrayliste)

```
    protected class Node {
        Node neste = null;
        T data;
        Node (T x) {
            data = x; }
    }
    protected Node start = null;
```

Node-objekter

T-objekter



Klassen Lenkeliste

- Vi implementerer samme interface **Liste** som **Arrayliste** implementerte

- Vi har bestemt datastruktur: En sammenlenket kjede av **Node**-objekter, og en referanse **start** til første Node-objekt

- Hvordan legge dette inn i klassen **Lenkeliste**?

- Vi deklarerer en *indre klasse* **Node** inne i klassen **Lenkeliste**

Indre klasser

- Klasser kan deklareres inne i andre klasser: Indre (eller nøstede) klasser.

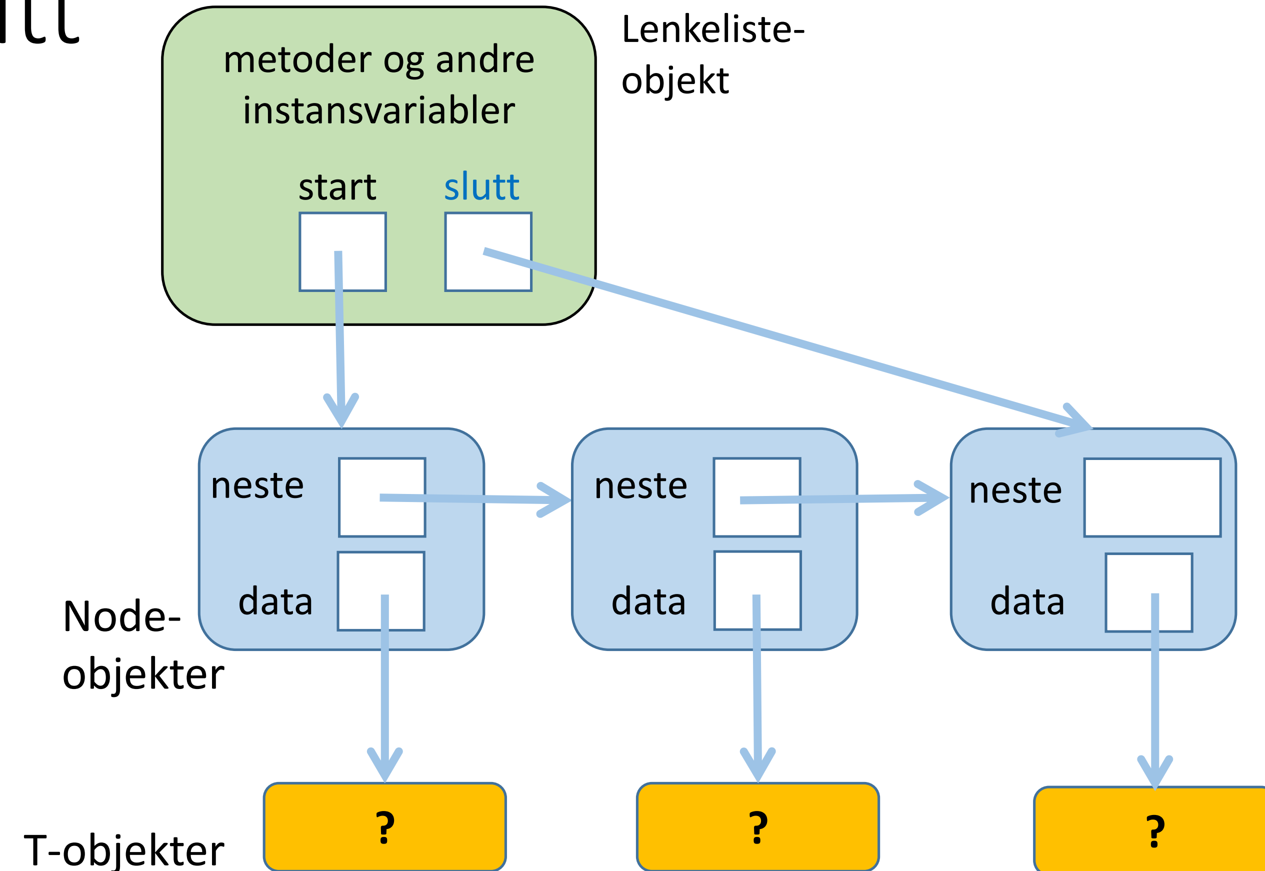
- Tydeliggjør at den kun brukes internt, og hindrer aksess fra utsiden (om vi deklarerer den private/ protected).

- Den indre klassen får en egen .class-fil ved kompilering (**Lenkeliste$Node.class**)

(det finnes også *statiske indre klasser* – det bruker vi ikke i IN1010)

Klassen Lenkeliste: Datastruktur og grensesnitt

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

    // Resten av grensesnittet
```

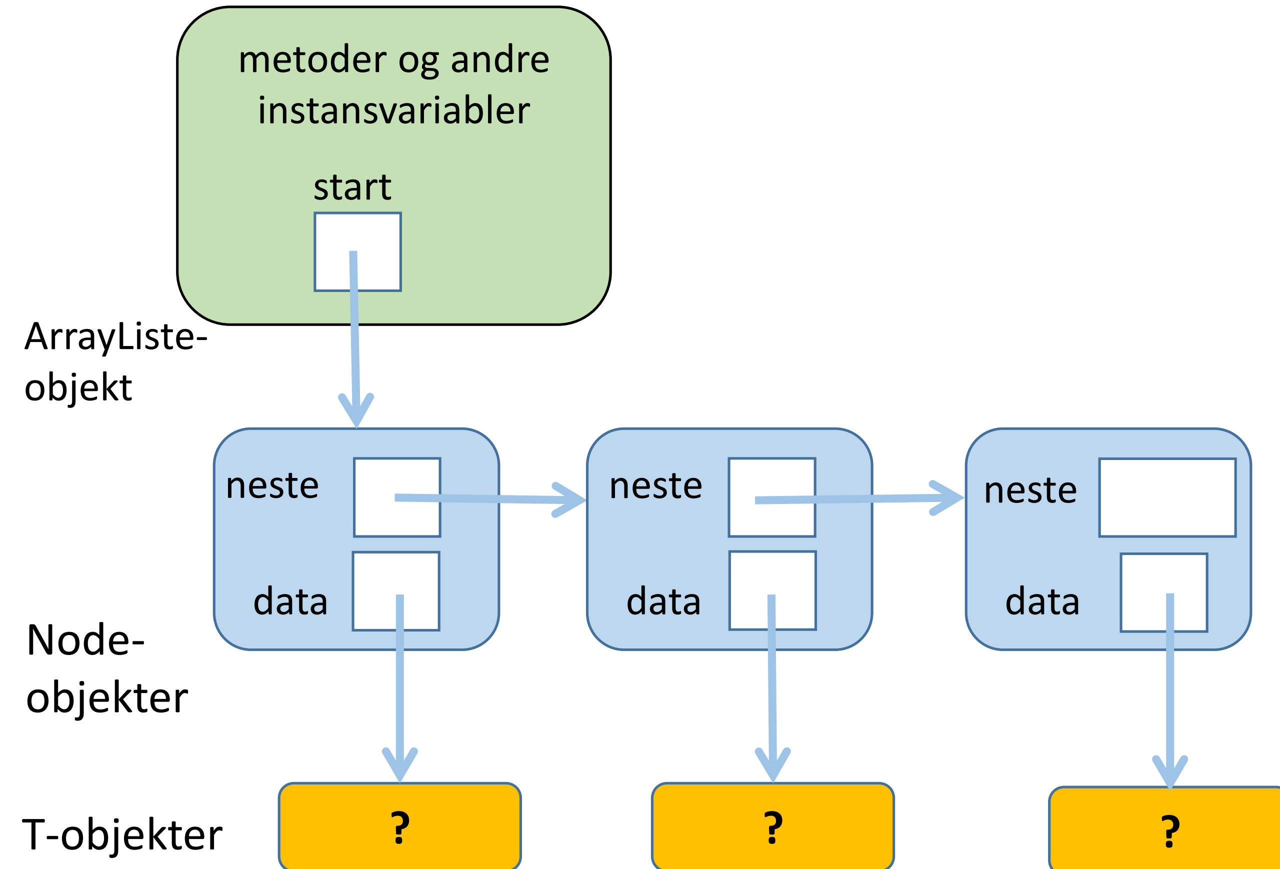


- Ofte nyttig med referanse til siste element. Mer neste uke, utelates her.

Invarianter for vår Lenkeliste

Gjelder alltid *mellom metodekall*

- **start** refererer alltid til Node-objektet med forste dataelement
- .. men hvis listen tom er **start == null**
- i Node-objektet som refererer til siste dataelement er alltid **neste** == **null**

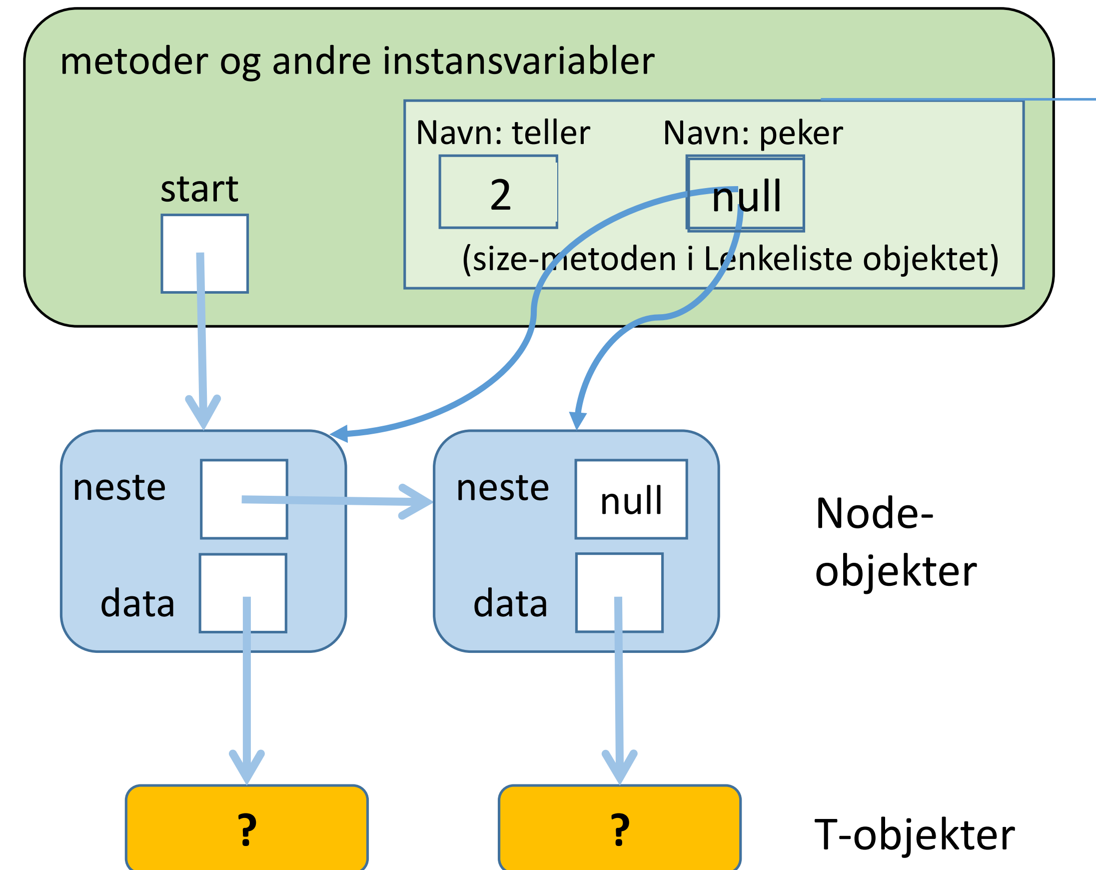


Hvordan finne størrelsen?

- Går gjennom liste og teller noder
- Trenger en referanse som flyttes gjennom listen

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

Forslag til et bedre alternativ?
Hvorfor bedre?



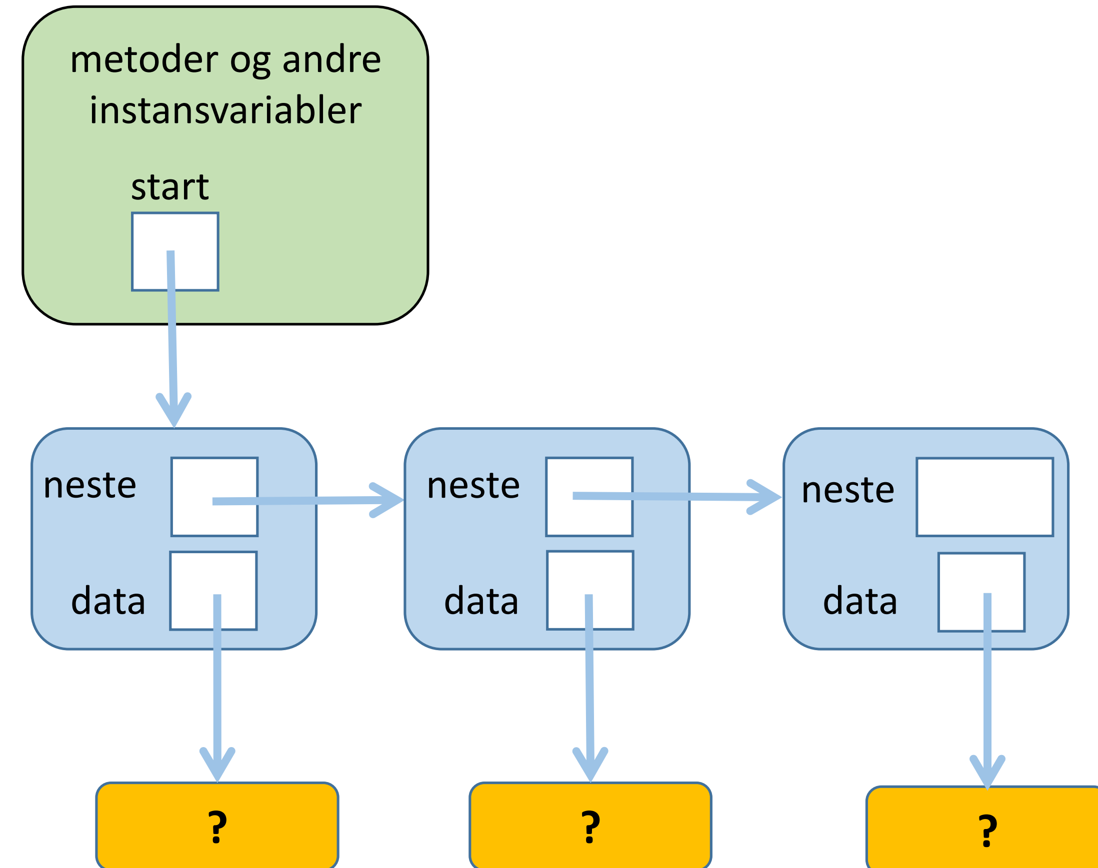
Hvordan hente et element?

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

- Går gjennom liste, teller oss frem til rett plass

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

NB: Hva skal vi returnere?



Hvordan fjerne et element fra listen?

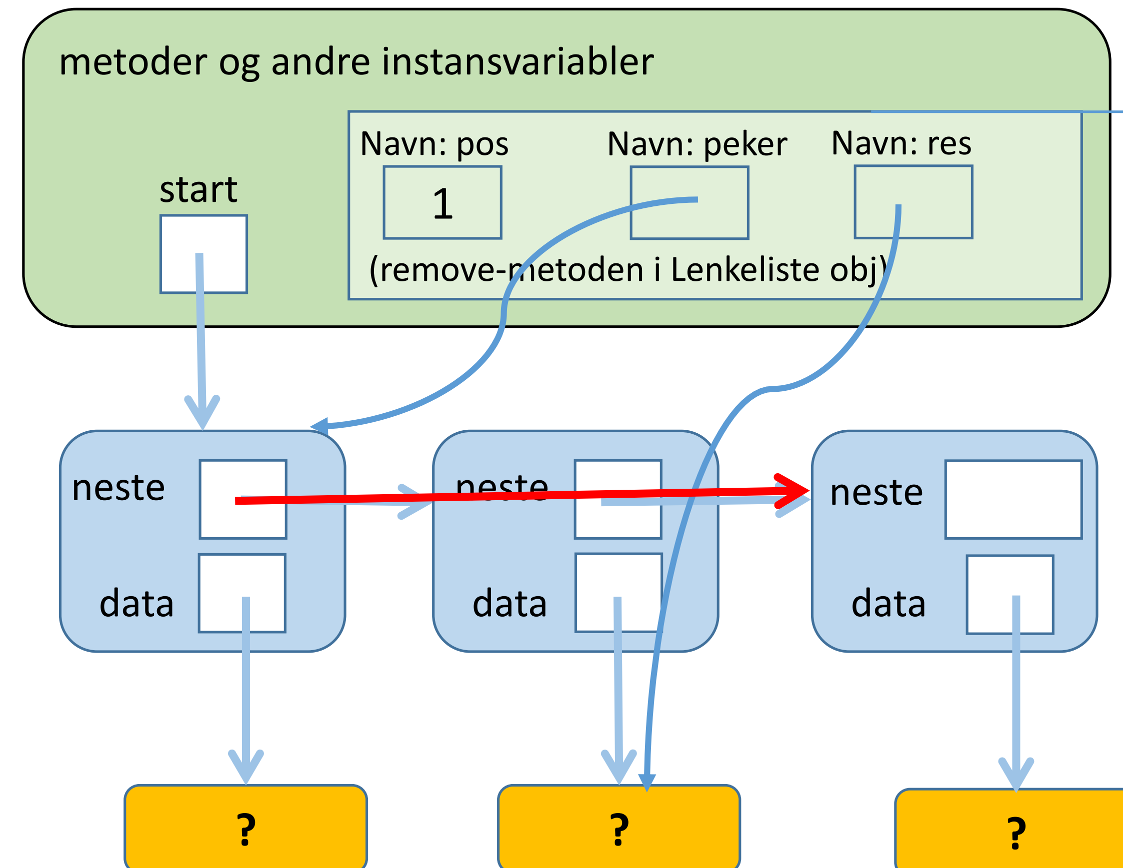
```
T remove(int pos);
```

- Teller oss frem til rett sted:

```
Node peker = start;
for (int i=0; i<pos-1; i++) {
    peker = peker.neste;
}
res = peker.neste.data;
peker.neste = peker.neste.neste;
```

- Hvilket element må vi stoppe på?
- Hvilken type er **res**?
- Hvilket spesialtilfelle må håndteres her?

Svar: Elementet *før* det som skal fjernes



Bruk av lenkelisten

Hvordan skiller et testprogram for lenkeliste-klassen seg fra testprogrammet vi skrev for arrayliste-klassen?

```
class TestArrayliste {
    public static void main(String[] args) {
        Liste<String> lx = new Arrayliste<>();
        // ....Sett inn 13 elementer, andre tester....

        for (int i = 0; i <= 12; i++)
            lx.add("A"+i);

        // Sjekk størrelsen:
        System.out.println("Listen har " + lx.size() + " elementer");

        // Marker element nr 10:
        lx.set(10, lx.get(10)+"*");

        // Fjern det første elementet:
        String s = lx.remove(0);
        System.out.println("Fjernet " + s);

        // Skriv ut innholdet:
        for (int i = 0; i < lx.size(); i++)
            System.out.println("Element " + i + ": " + lx.get(i));

        // Lag en feil:
        try {
            lx.remove(999);
        } catch (UlovligListeindeks u) {
            System.out.println("Feil: "+u.getMessage());
        }

        System.out.println("Fortsetter etter catch-blokken");

    }
}
```

Oppsummering

- Beholder: Hva og hvordan
  - Liste-interface
  - Implementering av Liste med array
  - Implementering av Liste med lenkeliste (kun utvalgte deler av koden)

- Nytt i Java
  - Klasseparametere og "generics"
  - Indre klasser

  Repetisjon/ eksempler:
  - Interface og arv, klassediagrammer
  - Egne Exceptions: Deklarasjon, opprettelse og behandling

Neste uke

- Ulike måter å implementere lenkelister
- Varianter av liste-grensesnitt:
  - kø (First In First Out – FIFO)
  - stabel (stack, Last In First Out – LIFO)
  - prioritetskø

- Mer Java
  - Innpakking ("autoboxing/ unboxing")
  - Å sammenligne objekter (interface Comparable)
  - Å gå gjennom alle elementer i en samling (Iterator)