Beholdere 1

- IN1010
- Uke 6 (i IN1010)
- Tirsdag 24. februar 2026



Dagens forelesning

Plan

- Beholdere
  - Motivasjon
  - Klassehierarki for beholdere
  - Lister
- Interface `StrengListe`
  - Implementasjon med array
  - Implementasjon med enkeltlenket liste
- Testing og unntakshåndtering
- Generiske typer
  - Interface `Liste`

I IN1010 forventer vi at du har dyp kjennskap til arrayer og lister

- Det vil si at du både kan *bruke* og *lage* dine egne liste-beholdere

Vi forventer at du kjenner til, og kan bruke, mengder og ordbøker

- (Hvordan de implementeres er tema for IN2010)

Introduksjon



Beholder

En datastruktur som brukes til å oppbevare en samling elementer

- Lister, mengder og ordbøker er eksempler på beholdere dere kjenner fra før
- Gitt en beholder, vil man vanligvis ha metoder for å
  - Sette in, hente ut (lese), fjerne, og spørre om antall elementer

- Beholdere kommer i svært mange varianter som støtter ulike operasjoner
  - Beholdere med samme grensesnitt kan ha hver sin implementasjon

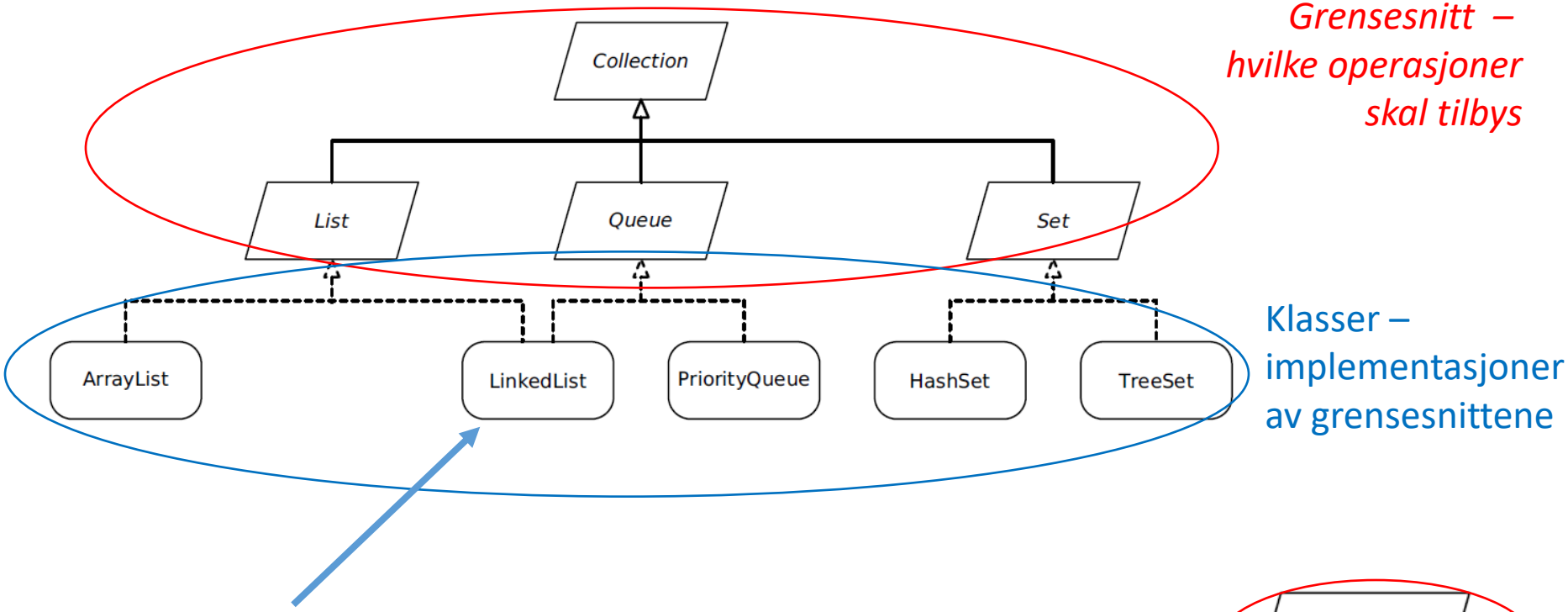
- Som regel er beholdere laget generiske
  - Det vil si at de kan holde på objekter av en hvilken som helst (men samme) type
  - I første omgang ser vi på beholdere der elementene er strenger

Hvorfor lære om beholdere?

- De er uunnværlige i svært mange programmer uansett språk
- Kunnskap gir bedre valg av verktøy i programmer du skriver
- Du kan få behov for å lage en skreddersydd beholder

- Beholdere er gode eksempler på objektorientert programmering
- Implementering gir nyttig trening i mekanismer som brukes ofte i
  - Java og objektorientering
  - … og i veldig mange andre språk

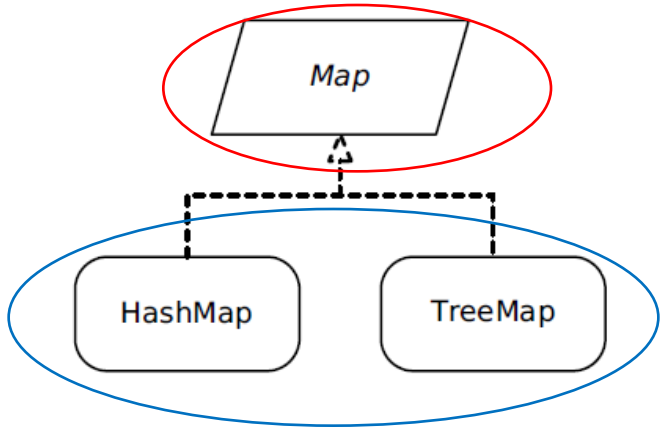
Java Collections Framework (utdrag! fra Big Java)



Grensesnitt – hvilke operasjoner skal tilbys

Klasser – implementasjoner av grensesnittene

Klassen LinkedList implementerer (her) 2 grensesnitt



Lister

En liste er en samling elementer av endelig størrelse i en bestemt ordning

- I en liste vil vi typisk kunne:
  - Hente ut størrelsen på listen
  - Sette inn, fjerne og hente ut (lese) et element først, sist, eller angitt med indeks
  - Iterere over elementene i listen i rekkefølgen som er bestemt av listen

- Java sitt grensesnitt `List` tilbyr dette (og mye mer)
- Vi skal definere et eget `StrengListe` interface som illustrerer typiske operasjoner, og implementere dette på ulike måter

Et forenklet grensesnitt for lister av *strenger*

```
interface StrengListe {
    int antall();

    void settInnFørst(String elem);

    void settInnSist(String elem);

    void fjernFørste();

    void fjernSiste();

    String hentFørste();

    String hentSiste();

    String hent(int indeks);
}
```

Implementasjon av beholdere

- Samme grensesnitt kan implementeres på flere måter
- En sentral beslutning i implementasjonen er valg av datastruktur
  - Hvordan kan vi lagre et vilkårlig antall elementer i beholderen?
- Vi skal se på to ulike tilnærminger:
  - Dynamiske arrayer
  - Lenkede lister (som det finnes mange variasjoner av)

Dynamiske arrayer



Dynamiske arrayer

- Et vanlig array (som i Java) opprettes med en *fast størrelse*
- Et dynamisk array *utvides automatisk* ved behov
  - Men automatikken må implementeres!

- `ArrayList` i Java er et dynamisk array
  - Implementerer grensesnittet `List`

- Vi skal implementere grensesnittet `StrengListe` som et dynamisk array

Klassedeklarasjon `DynamiskStrengArray`

- Elementene lagres i et array, fra posisjon 0
- `antall` holder rede på hvor mange plasser som er i bruk
- HVIS arrayet ikke er fullt viser `antall` også hva som er neste ledige posisjon

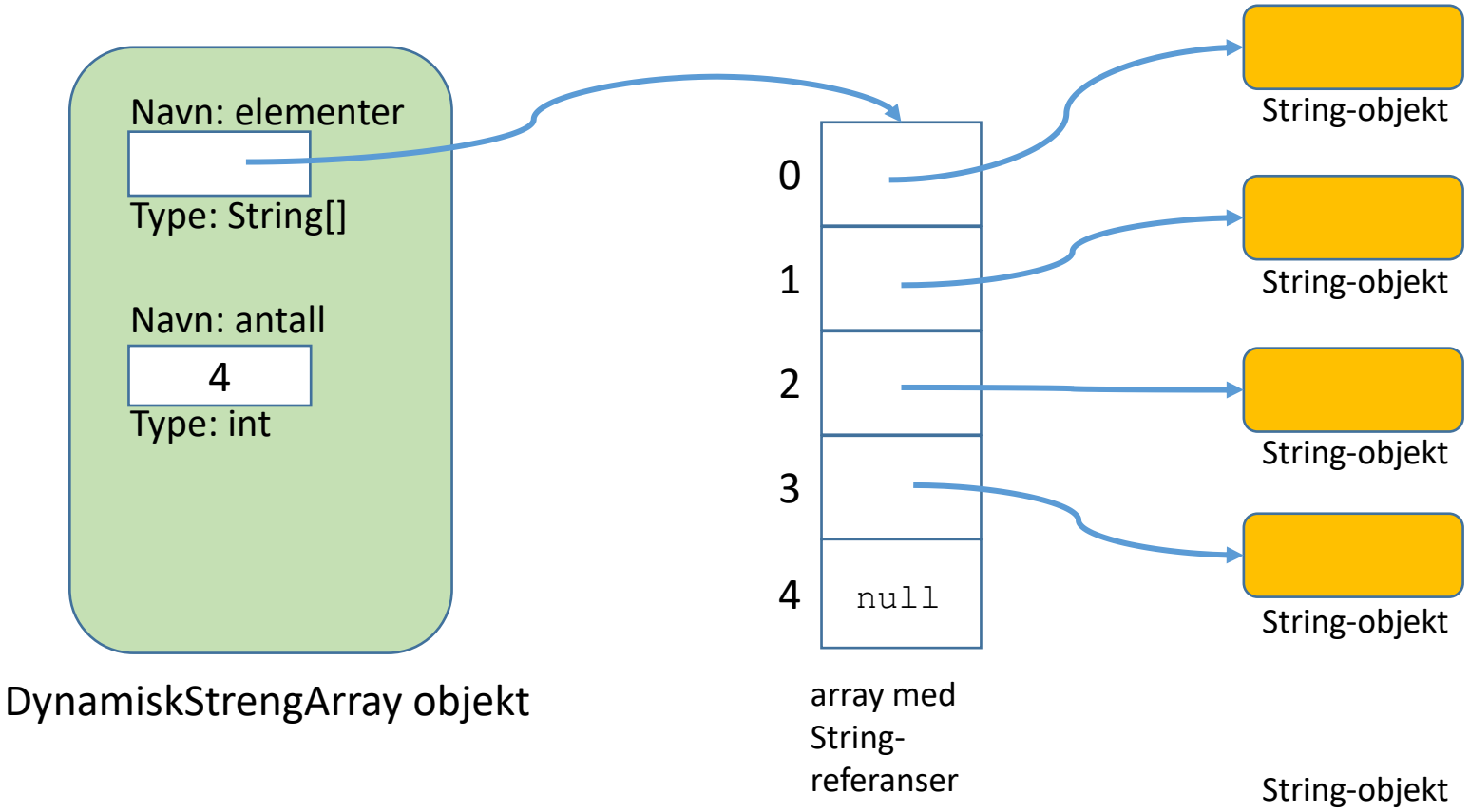
```
class DynamiskStrengArray implements StrengListe {


    private int antall;
    private String[] elementer;


    DynamiskStrengArray() {
        antall = 0;
        elementer = new String[5];
    }

   // metoder
```

DynamiskStrengArray objekt
(her med 4 elementer)



Finne antall i DynamiskStrengArray

- Må vite hvor mange plasser som er i bruk:
  I alle metoder som modifiserer listen må vi sørge for å oppdatere antall
- Merk at en metode og en variabel kan samme navn uten at det skaper problemer

```
    @Override
    public int antall() {
        return antall;
    }
```

- `@Override` er av plasshensyn ikke tatt med i senere utdrag
- Men bør alltid være med (når vi implementerer eller overskriver metoder) selv om Java ikke krever det
- Sikrer feilmelding hvis metode med samme signatur *ikke* finnes i superklasse/ interface

Innsetting på slutten

- Før vi setter inn et element, må vi sikre at vi har kapasitet (arrayen er jo begrenset)
  - Det vil si vi må sørge for at det er plass til *minst* ett element til
  - Vi antar at vi har en metode `sikreKapasitet()` som sørger for dette
- Vi setter verdien på posisjon `antall` til å være strengen gitt som argument
- Til slutt inkrementerer vi `antall`

```
    public void settInnSist(String elem) {
        sikreKapasitet();
        elementer[antall] = elem;
        antall++;
    }
```

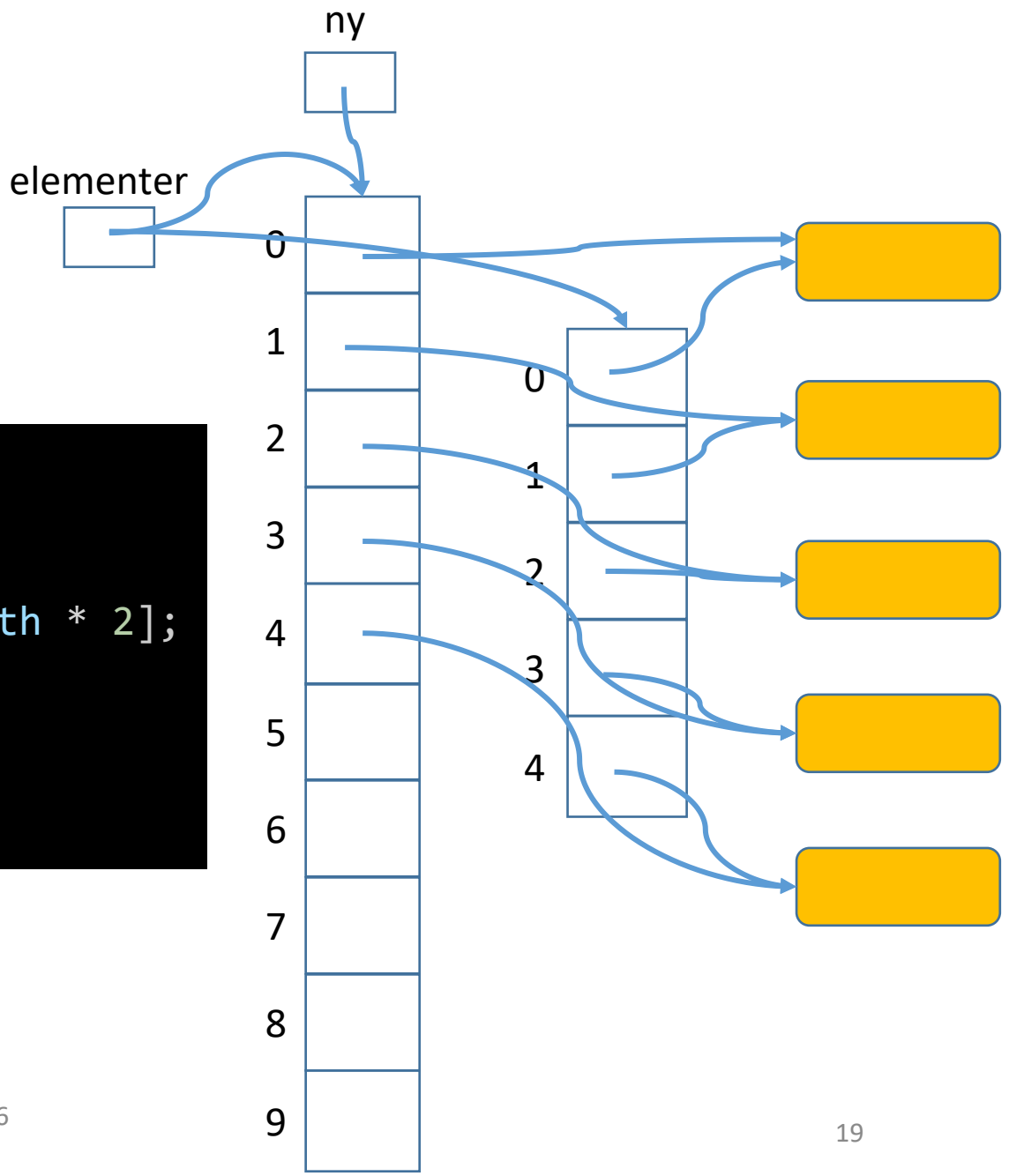
Sikre kapasitet

- Dersom `antall` er mindre enn `elementer.length` gjør vi ingenting
- Ellers oppretter vi et nytt array med dobbelt så mye plass
  - Det nye arrayet fylles opp med verdiene fra `elementer`
- Til slutt setter vi `elementer` til å peke på det nye arrayet
  - Når `sikreKapasitet` returnerer har vi plass til flere elementer!

```
    private void sikreKapasitet() {
        if (antall < elementer.length) {
            return;
        }
        String[] ny = new String[elementer.length * 2];
        for (int i = 0; i < antall; i++) {
            ny[i] = elementer[i];
        }
        elementer = ny;
    }
```

Hva skjer ved fullt array? (animasjon)

```
    private void sikreKapasitet() {
        if (antall < elementer.length) {
            return;
        }
        String[] ny = new String[elementer.length * 2];
        for (int i = 0; i < antall; i++) {
            ny[i] = elementer[i];
        }
        elementer = ny;
    }
```



Sletting på slutten

- Når vi fjerner siste element må vi huske å sjekke at `antall` ikke er 0
  - I så fall gjør vi ingenting
  - Alternativt kunne vi kastet et passende unntak
- Vi senker `antall` og setter elementet på posisjon `antall` til `null`

```
    public void fjernSiste() {
        if (antall == 0) {
            return;
        }
        antall--;
        elementer[antall] = null;
    }
```

Hente første og siste

- Å hente første er å slå opp på indeks 0
- Å hente siste er å slå opp på indeks `antall - 1`
- Vi returnerer `null` dersom arrayet er tomt
  - Her kunne vi også kastet et passende unntak

```
    public String hentFørste() {
        if (antall == 0) {
            return null;
        }
        return elementer[0];
    }
```

```
    public String hentSiste() {
        if (antall == 0) {
            return null;
        }
        return elementer[antall - 1];
    }
```

Hente på indeks

- Å hente på indeks gjøres ved å slå opp på indeksen i elementer
- Her kaster vi `IndexOutOfBoundsException` dersom indeksen er negativ eller større (eller lik) `antall`.

```
    public String hent(int indeks) {
        if (indeks < 0 || antall() <= indeks) {
            throw new IndexOutOfBoundsException();
        }
        return elementer[indeks];
    }
```

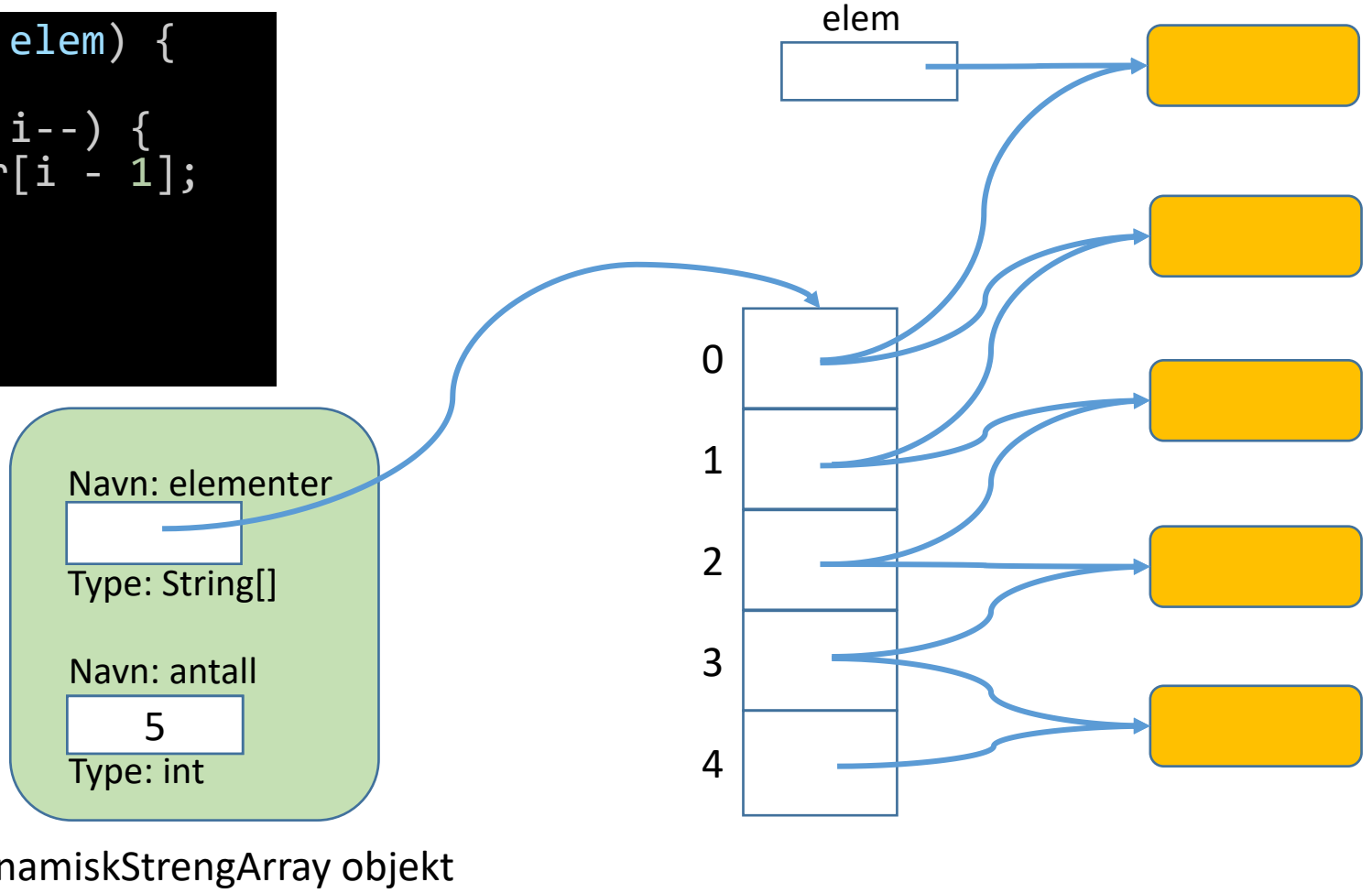
Sett inn først

- Innsetting først i et dynamisk array er litt mer komplisert
- Antakeligvis ligger det allerede noe på posisjon 0
- Da må vi flytte alle elementer en posisjon frem

Sett inn først (animasjon)

```
public void settInnFørst(String elem) {
    sikreKapasitet();
    for (int i = antall; 0 < i; i--) {
        elementer[i] = elementer[i - 1];
    }
    elementer[0] = elem;
    antall++;
}
```

Hva skjer om beholderen er tom?



DynamiskStrengArray objekt

Fjern først

- Tilsvarende krever fjerning først litt rydding (dessuten en sjekk på at noe fins)
- Her må vi flytte alle gjenværende elementer en posisjon tilbake
- Vi må også huske å sette siste posisjon til null

```
public void fjernFørste() {
        if (antall == 0) {
            return;
        }
        for (int i = 0; i < antall - 1; i++) {
            elementer[i] = elementer[i + 1];
        }
        antall--;
        elementer[antall] = null;
    }
```

Invarianter

- 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 `DynamiskStrengArray`

- Har vi identifisert noen invarianter allerede?
  - `antall` angir hvor mange elementer beholderen har
  - arrayen `elementer` er alltid fyllt fra indeks `0` til `antall-1`
  - `antall == elementer.length` betyr at arrayen er full!
    - ellers er `antall` indeks for neste ledige plass

- Nyttig å formulere mens man designer datastrukturen, og bruke når man skriver metodene

Ufullstendig testprogram for `StrengListe` beholdere

```
interface StrengListe {
 √  int antall();

 √  void settInnFørst(String elem);

 √  void settInnSist(String elem);

 √  void fjernFørste();

 √  void fjernSiste();

    String hentFørste();

    String hentSiste();

  √ String hent(int indeks);
}
```

- Setter inn 13 elementer sist
- Setter inn 1 element først
- Skriver ut antall
- Skriver ut listen
- Fjerner første element
- Fjerner siste element
- Skriver ut listen

```java
public class TestStrengListe {
    public static void main(String[] args) {
        StrengListe sListe = new DynamiskStrengArray();

        System.out.println();
        for (int i = 0; i < 13; i++) {
            sListe.settInnSist("A" + i);
        }
        sListe.settInnFørst("XTRA");

        System.out.println("Listen har " + sListe.antall() + " elementer");
        for (int i = 0; i < sListe.antall(); i++) {
            System.out.println("Element " + i + ": " + sListe.hent(i));
        }
        System.out.println();

        sListe.fjernFørste();
        sListe.fjernSiste();

        for (int i = 0; i < sListe.antall(); i++) {
            System.out.println("Element " + i + ": " + sListe.hent(i));
        }
    }
}
```

```
Listen har 14 elementer
Element 0: XTRA
Element 1: A0
Element 2: A1
Element 3: A2
Element 4: A3
Element 5: A4
Element 6: A5
Element 7: A6
Element 8: A7
Element 9: A8
Element 10: A9
Element 11: A10
Element 12: A11
Element 13: A12

Element 0: A0
Element 1: A1
Element 2: A2
Element 3: A3
Element 4: A4
Element 5: A5
Element 6: A6
Element 7: A7
Element 8: A8
Element 9: A9
Element 10: A10
Element 11: A11
```

Lenkede lister

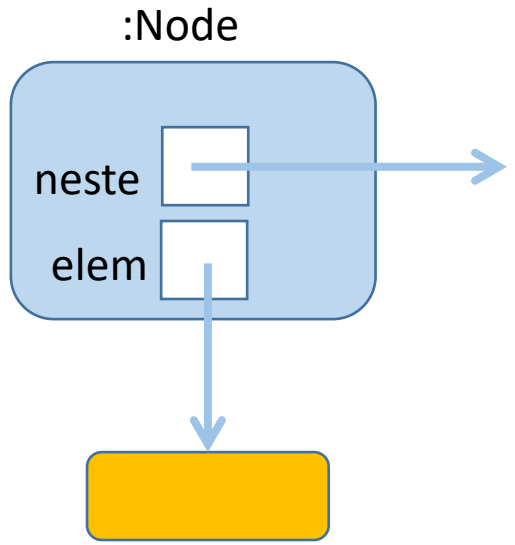


En annen implementasjon av StrengListe

- Kan vi lage en beholder der vi *alltid* kan ta inn ett element til uten å "bygge om" datastrukturen vår?
  - Vi trenger en datastruktur der det *alltid er en ledig referanse* til et nytt elementet

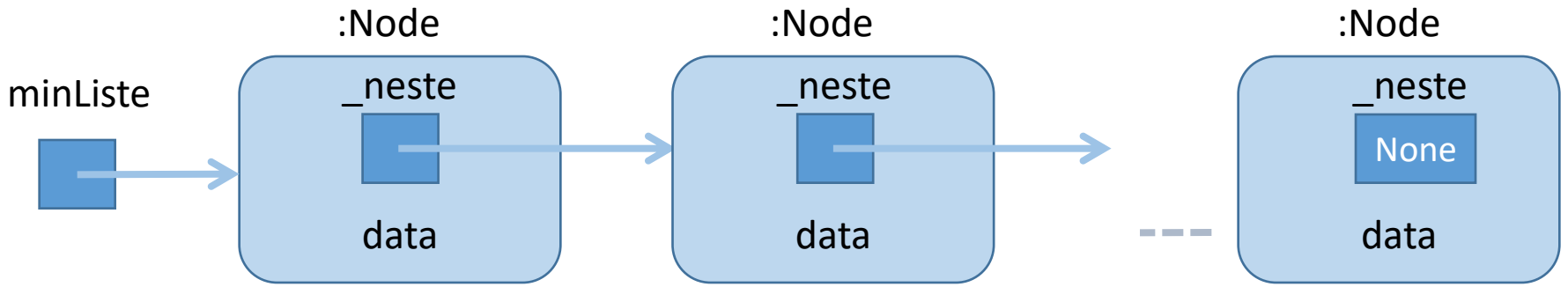
Vi bruker *noder* for å få til dette

- for hvert element, oppretter vi et hjelpe-objekt (en node)
- .. som skal peke på (referere til) det nye elementet –
- .. OG kan peke på et nytt hjelpeobjekt (node)



- Beholderklassen trenger kun én peker (referanse): Til den første noden

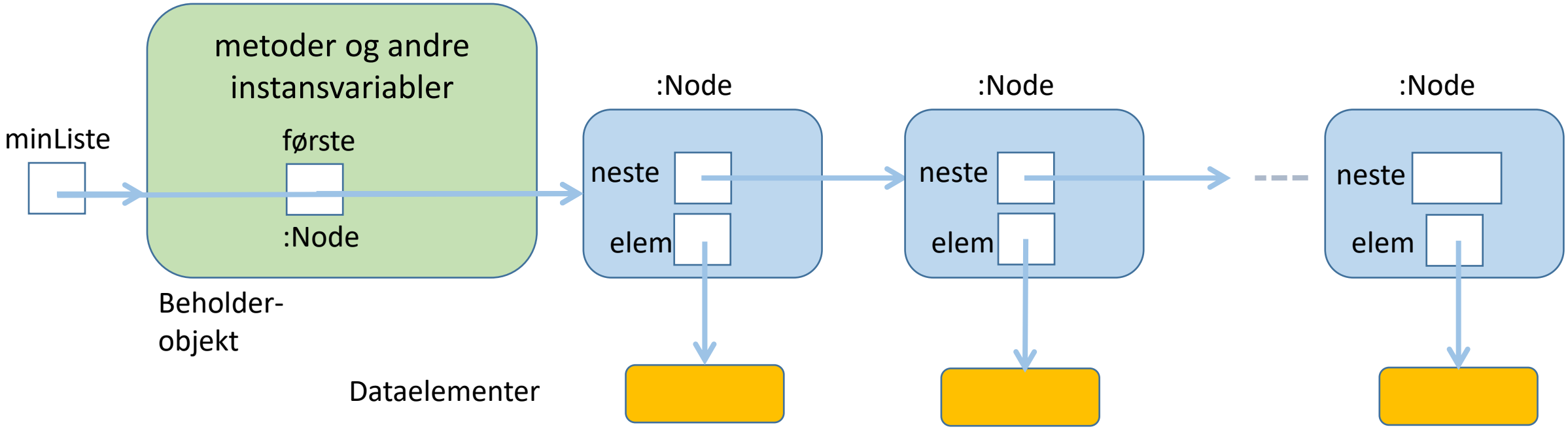
Lenkeliste (NB: figur fra IN1000, Python)



- Poenget med en lenkeliste er at for hvert nye Node-objekt vi lager – så får vi samtidig en referansevariabel som kan referere til enda et nytt objekt
- dvs hver node må kunne referere til en annen node ved hjelp av en nestepeker
- dermed får vi en lenket liste av noder – og trenger bare ha én referanse, til den første noden

Beholder med 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



Klassen `EnkeltLenketStrengListe`

- Vi implementerer samme interface `StrengListe` som `DynamiskStrengArray` implementerte

- Vi har bestemt datastruktur: En sammenlenket kjede av Node-objekter, og en peker `første` til første Node-objekt (eller `null` om beholderen er tom)

- Hvordan legge dette inn i klassen `EnkeltLenketStrengListe`?

- Vi deklarerer en *indre klasse* `Node` inne i klassen `EnkeltLenketStrengListe`

Indre klasser

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

- Tydeliggjør at den kun brukes internt, og hindrer aksess fra utsiden om vi deklarerer den `private` eller `protected`.
  (`protected` hvis den skal aksesseres fra subklasser)

- 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)

Klassedeklarasjon – beholder med enkeltlenket liste

Vi lar Node være en *indre klasse*

- Her settes neste-pekeren i konstruktøren til `Node`
  (i `StrengListe` er `neste` alltid `første` eller `null` for en ny node)
- Ofte har man kun elementet som parameter til `Node`-konstruktøren, og finner plassering etter at noden er laget

```
class EnkeltLenketStrengListe implements StrengListe {
    private class Node {
        String elem;
        Node neste;

        Node(String elem, Node neste) {
            this.elem = elem;
            this.neste = neste;
        }
    }
    private Node første;

    // Resten av klassen
}
```

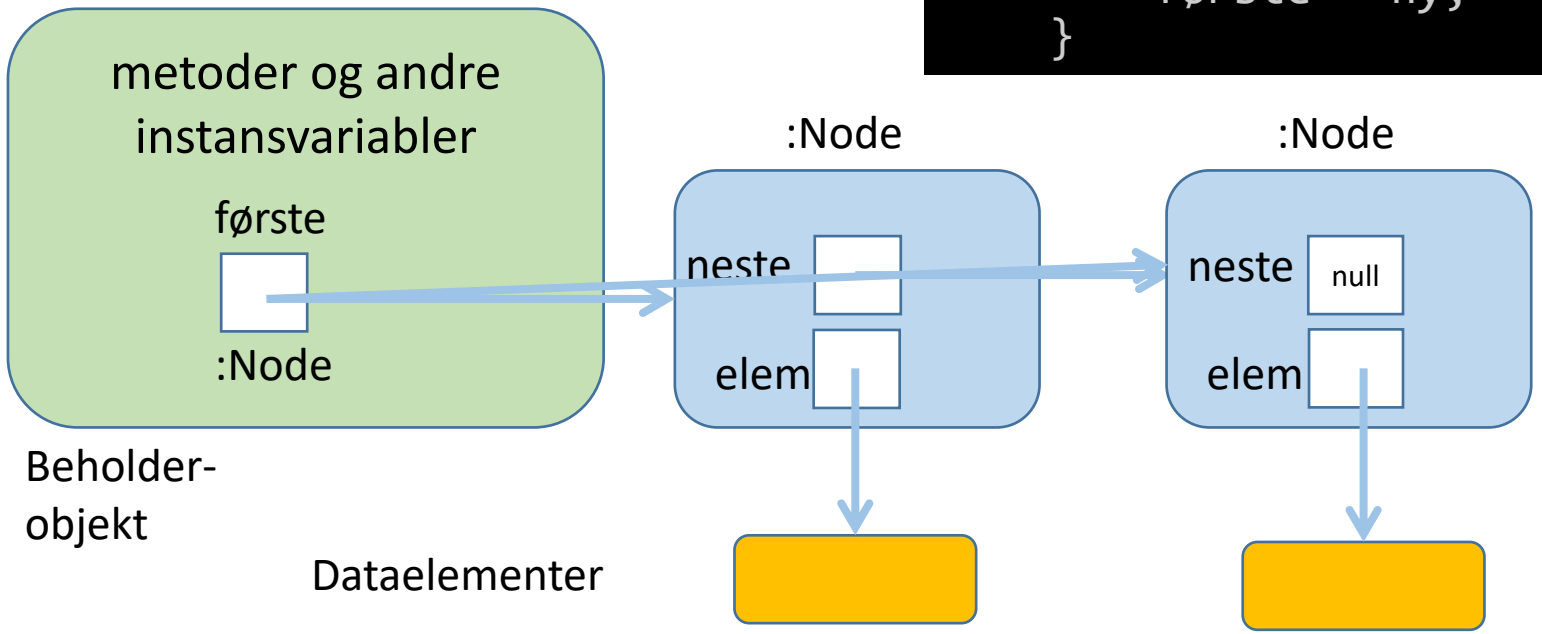
Sett inn først i listen

- Lager en ny node for det nye elementet
- Lar den nye noden sin `neste` peke på det beholderens `første` peker på (argument til `Node`-konstruktøren)
- Setter beholderens `første` til å peke på den nye noden

```
    public void settInnFørst(String elem) {
        Node ny = new Node(elem, første);
        første = ny;
    }
```

Sett inn først (animasjon)

```
public void settInnFørst(String elem) {
    Node ny = new Node(elem, første);
    første = ny;
}
```




```
Node(String elem, Node neste) {
    this.elem = elem;
    this.neste = neste;
```

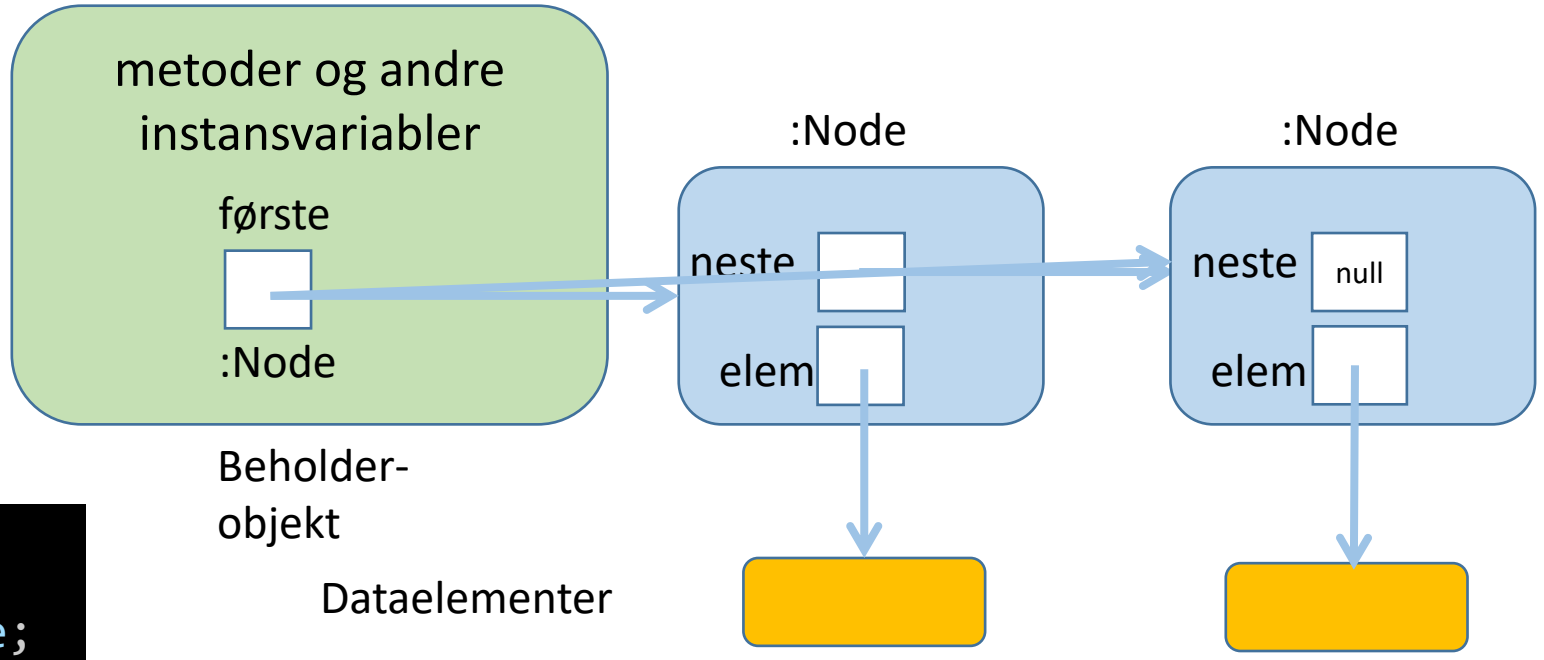
Fjern første element i listen

- For å fjerne første element må vi sette `første` til å peke på `første.neste`
- Nå er det ikke lenger noe som peker på noden `første` pekte på
- Da «forsvinner» den ved neste søppeltømming (garbage collection)
- Igjen gjør vi ingenting dersom `første` er `null`

```
    public void fjernFørste() {
        if (første != null) {
            første = første.neste;
        }
    }
```

Fjern første element i listen (animasjon)

```
public void fjernFørste() {
    if (første != null) {
        første = første.neste;
    }
}
```



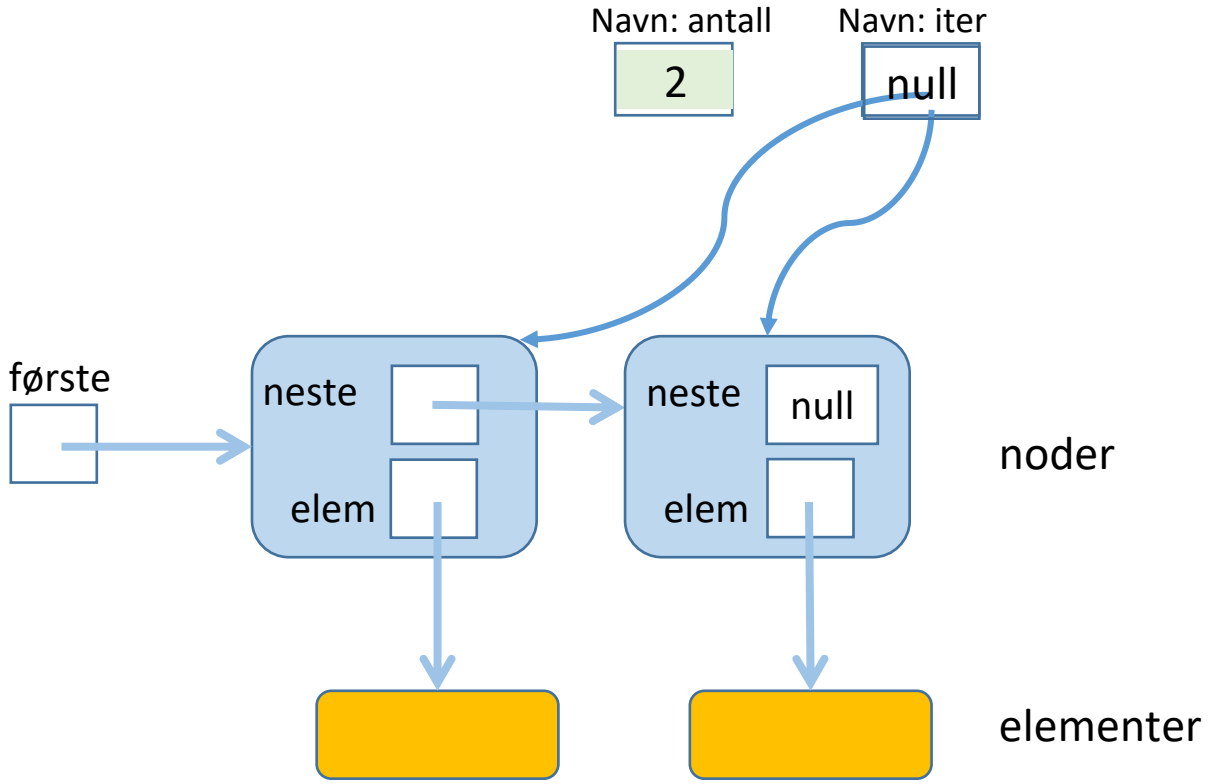
Finne antall

- Slik klassedeklarasjonen ser ut, må vi telle noder for å finne antallet
  - Vi oppretter en nodepeker `iter` som initialiseres til det samme som `første`
  - Deretter følge neste-pekere med `iter` frem til `iter` blir `null`
  - Vi følger en neste-peker ved å si `iter = iter.neste`
  - Oppdaterer `antall` for hver node `iter` peker på

```
    public int antall() {
        int antall = 0;
        Node iter = første;
        while (iter != null) {
            antall++;
            iter = iter.neste;
        }
        return antall;
    }
```

Finne antall (animasjon)

```
    public int antall() {
        int antall = 0;
        Node iter = første;
        while (iter != null) {
            antall++;
            iter = iter.neste;
        }
        return antall;
    }
```



Sett inn sist

- Flytter peker fra første node via `neste`-pekere til en node der `neste == null`

```
    public void settInnSist(String elem) {
        Node ny = new Node(elem, null);
        if (første == null) {
            første = ny;
            return;
        }
        Node iter = første;
        while (iter.neste != null) {
            iter = iter.neste;
        }
        iter.neste = ny;
    }
```

Sett inn sist med siste-peker
(alternativ datastruktur)

- Om vi legger til en `Node`-peker `siste` i beholderen (krever ny klassedefinisjon)
- .. slipper vi å iterere over alle nodene i listen for å sette inn bakerst

```
    public void settInnSist(String elem) {
        Node ny = new Node(elem, null);
        if (første == null) {
            første = ny;
            siste = ny;
        } else {
            siste.neste = ny;
            siste = ny;
        }
    }
```

Fjerne siste

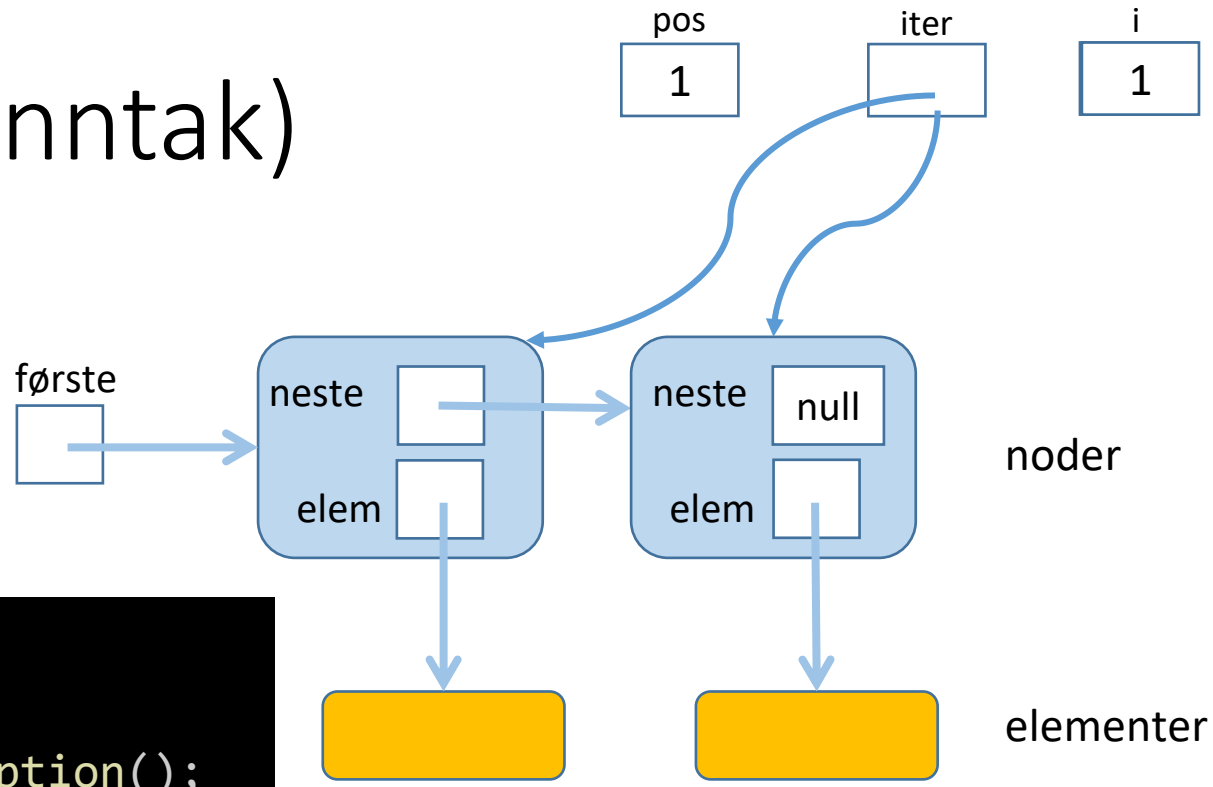
- For å fjerne den siste må vi sette `neste`-pekeren i *nest siste* node til `null`
- Altså trenger vi å lete oss frem til denne (ikke til `siste`)
- Nest siste node sin `neste` settes så til `null`

- Hvilke spesialtilfeller har vi her?
- Kunne vi forenklet fjerning med en siste-peker?

```
    public void fjernSiste() {
        if (første == null) {
            return;
        }
        if (første.neste == null) {
            første = null;
            return;
        }
        Node iter = første;
        while (iter.neste.neste != null) {
            iter = iter.neste;
        }
        iter.neste = null;
    }
```

Hente på indeks (med unntak)

- Sjekker om indeksen er gyldig
- Teller oss frem til riktig node
- Hva skal vi returnere?




```
    @Override
    public String hent(int pos) {
        if (pos < 0 || pos >= antall()) {
            throw new IndexOutOfBoundsException();
        }
        Node iter = første;
        for (int i = 0; i<pos; i++) {
            iter = iter.neste;
        }
        return iter.elem;
    }
```

Testing og unntakshåndtering



Enkelt testprogram for `StrengListe` beholdere (som vist før)

```
interface StrengListe {
    int antall();

    void settInnFørst(String elem);

    void settInnSist(String elem);

    void fjernFørste();

    void fjernSiste();

    String hentFørste();

    String hentSiste();

    String hent(int indeks);
}
```

- Setter inn 13 elementer sist
- Setter inn 1 element først
- Skriver ut antall
- Skriver ut listen
- Fjerner første element
- Fjerner siste element
- Skriver ut listen

```java
public class TestStrengListe {
    public static void main(String[] args) {
        StrengListe sListe = new EnkeltLenketStrengListe();

        System.out.println();
        for (int i = 0; i < 13; i++) {
            sListe.settInnSist("A" + i);
        }
        sListe.settInnFørst("XTRA");

        System.out.println("Listen har " + sListe.antall() + " elementer");
        for (int i = 0; i < sListe.antall(); i++) {
            System.out.println("Element " + i + ": " + sListe.hent(i));
        }
        System.out.println();

        sListe.fjernFørste();
        sListe.fjernSiste();

        for (int i = 0; i < sListe.antall(); i++) {
            System.out.println("Element " + i + ": " + sListe.hent(i));
        }
    }
}
```

```
Listen har 14 elementer
Element 0: XTRA
Element 1: A0
Element 2: A1
Element 3: A2
Element 4: A3
Element 5: A4
Element 6: A5
Element 7: A6
Element 8: A7
Element 9: A8
Element 10: A9
Element 11: A10
Element 12: A11
Element 13: A12

Element 0: A0
Element 1: A1
Element 2: A2
Element 3: A3
Element 4: A4
Element 5: A5
Element 6: A6
Element 7: A7
Element 8: A8
Element 9: A9
Element 10: A10
Element 11: A11
```

Mangler i testprogrammet

- Vi testet ikke metodene for alle mulige tilstander i beholderen
- Fanget ikke unntaket vi kastet i metoden `finn`

```java
public class TestStrengListeUnntak {
    public static void main(String[] args) {
        StrengListe sListe = new EnkeltLenketStrengListe();
        for (int i = 0; i < 3; i++) {
            sListe.settInnSist("A" + i);

        // Prøver å hente ut på ulovlig indeks
        System.out.println(sListe.hent(3));
    }
}
```

```
Exception in thread "main" java.lang.IndexOutOfBoundsException
        at EnkeltLenketStrengListe.hent(EnkeltLenketStrengListe.java:92)
        at TestStrengListeUnntak.main(TestStrengListeUnntak.java:16)
```

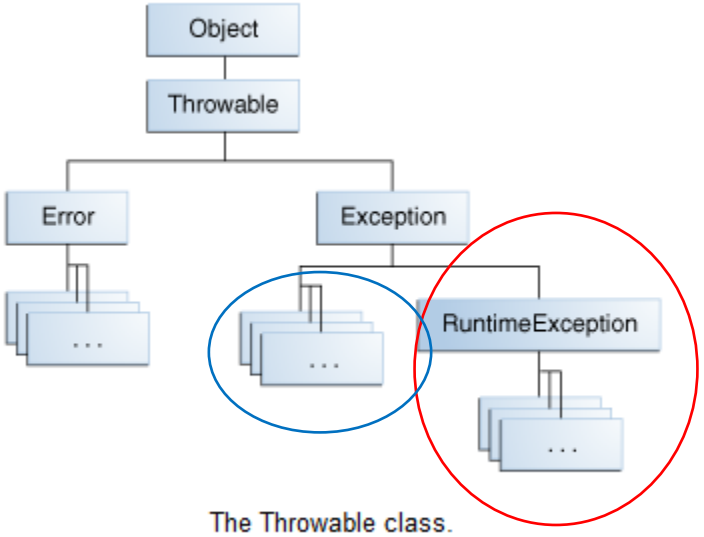
- `Kjøretidsfeil, MEN litt mer informativ enn NullPointerException` (som vi ellers ville fått)

Feilhåndtering

- Hva kan gå galt?
  - Feil i Java => Error, trenger ikke tenke på/ håndtere
  - Feil 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: IndexOutOfBoundsException)
  - du velger om du vil teste og eventuell 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

Det vi gjorde i hent-metoden

- 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 mer detaljer*
  - Om du *ikke* fanger en Exception som oppstår, håndterer Java den "på sin måte":
    Avslutter programmet med en feilmelding

Det vi gjorde i testprogrammet

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)

Egen-definert Exception som kunne vært brukt i hent-metoden

```
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 nedenfor` – NB annet eksempel, ny klasse)
- ... 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 `hent` eller `remove`) kan vi skriver 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

Generiske typer



Beholdere som ikke bare kan lagre strenger

- Interfacet `StrengListe` og klassene `DynamiskStrengArray` og `EnkeltLenketStrengListe` kan bare brukes til `String`-objekter
- Vanligvis ønsker vi beholdere som er mer *generiske*
  - der vi kan velge hva slags objekter den skal lagre *når vi instansierer (oppretter) beholderen*

- Det vil si slik `ArrayList` og `HashMap` fungerer i Java

```
public static void main(String[] args) {

    HashMap<String, Student> studentene = new HashMap<>();

    ArrayList<String> studentnavn = new ArrayList<>();

}
```

Generiske klasser og grensesnitt

- Vi angir at klassen eller grensesnittet skal ta en eller flere *typeparametere*
- Må angi en klasse eller et interface

```
interface StrengListe {/*… */}
```



```
interface Liste<T> {/*… */}
```

```
class EnkeltLenketStrengListe implements StrengListe {/*… */}
```



```
class EnkeltLenketListe<T> implements Liste<T> {/*… */}
```

Generiske klasser og grensesnitt

- Vi angir at klassen eller grensesnittet skal ta en eller flere *typeparametere*

```
interface StrengListe {/*… */}
```



```
interface Liste<T> {/*… */}
```

- T er en typeparameter til grensesnittet Liste
- I Java er det en konvensjon at typeparametere består av én stor bokstav, lett å kjenne igjen
- Omsluttes av spissparanteser <> i motsetning til metodeparametere som omsluttes av ()

- Vi kan definere beholderklassene med typeparameter på tilsvarende måte

```
class EnkeltLenketStrengListe implements StrengListe {/*… */}
```



```
class EnkeltLenketListe<T> implements Liste<T> {/*… */}
```

Definisjon av generisk Liste-grensesnitt

- Vi kan nå definere et generisk grensesnitt `Liste` med en typeparameter

- Og bruker typeparameteren alle steder der vi oppga typen `String` i `StringListe`

```
interface Liste<T> {
    int antall();

    void settInnFørst(T elem);

    void settInnSist(T elem);

    void fjernFørste();

    void fjernSiste();

    T hentFørste();

    T hentSiste();

    T hent(int indeks);
}
```

Definisjon av generisk klasse (utdrag)

- Utdrag av ny klasse `EnkeltLenketListe`
- `String` erstattes av typeparameteren `T`
- Merk at vi (her også) må holde rede på når vi refererer til *noder* og når vi refererer til *elementer*

```
class EnkeltLenketListe<T> implements Liste<T> {

    private class Node {
        T elem;
        Node neste;

        Node(T elem, Node neste) {
            this.elem = elem;
            this.neste = neste;
        }
    }

    private Node første;
```

Instansiering og bruk av generiske klasser

- En generisk klasse eller grensesnitt må instansieres med et typeargument

```
Liste<String> liste = new EnkeltLenketListe<>();

liste.settInnFørst("Hei");
String s = liste.hentFørste();
```

- Her er `String` typeargument til `Liste` ved deklarasjon av peker-variabelen
  - Typeargumentet blir implisitt gitt til `EnkeltLenketListe` ved <>

- Kompilatoren sjekker at elementer som settes inn og hentes ut av listen er riktig type
- (vi skal se på hvordan vi kan lage beholdere for primitive typer neste uke)

Straks kvelden!



Innleveringsoppgave 3

- Start denne uken!
  - Oppgave 6 kan vente til neste uke

- God og relevant trening i:
  - Ting dere har lært før
    (Java, programmering, arv, polymorfi, interface, unntak, referanser)
  - Nytt stoff
    (generiske typer, beholdere med ulike grensesnitt og datastrukturer, grensesnittet Comparable som gjennomgås neste uke)

- Du får gjøre egne valg av datastruktur og fremgangsmåte
  - Mange alternativer som fungerer
  - Prøv å tenke gjennom alle situasjoner (ingen/ ett/ flere elementer, først/ sist/ midt i listen, …)
  - Prøv gjerne flere løsninger med ulik datastruktur

Oppsummering

- Beholdere er gode eksempler på typehierarkier og gir nyttig trening i bruk av interface, subklasser og mange andre Java mekanismer:
  - Hvordan ett grensesnitt kan implementeres på flere måter
  - Hvordan vi kan lage egne dynamiske arrayer
  - Hvordan lenkede lister kan bygges opp og brukes

- Feil/ unntakshåndtering med og uten Exceptions
  - Vi returnerte bare `null` i `hentSiste`- og `hentFørste`-metodene
  - I `hent` kastet vi et unntak som identifiserte feilen, men ga ikke ekstra informasjon
  - I `remove` laget vi et eget unntak med ekstra info, og fanget den med egen feilmelding

- Vi har laget et generisk grensesnitt og beholder ved hjelp av typeparametere – mer om dette neste uke