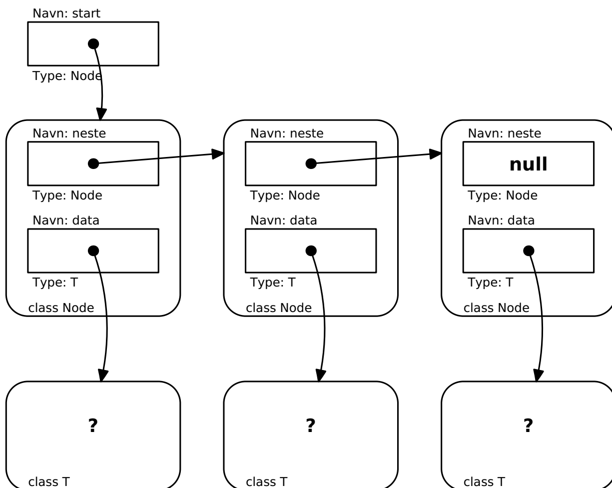
Dagens tema

- Litt mer om vanlige lister
  - Enveis- og toveislister
  - Innkapsling («boxing») (Big Java 6.8.5)
- Nyttige varianter av lister:
  - Stabler («stacks») (Big Java 15.5.1)
  - Køer («queues») (Big Java 15.5.2)
  - Prioritetskøer («priority queues») (Big Java 15.5.3)
    - Sammenligning av objekter (Comparable) (Big Java 9.6.3)
- Hvordan gå elegant gjennom en samling data
  - Iterator-er (Big Java 15.2)

Repetisjon: Enveislister



Hva er bra og mindre bra med slike lister?

Enkle å programmere

Fleksible

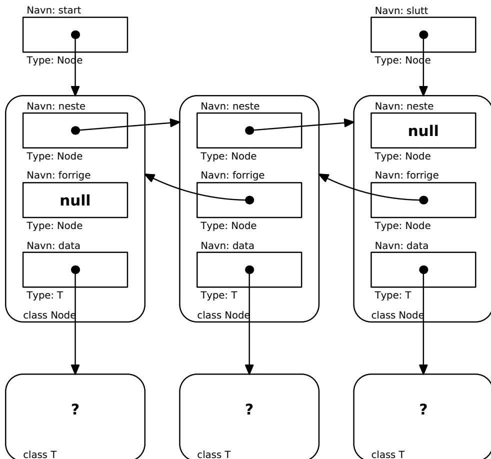
Kan av og til bli mye leting:

- Når vi skal finne et element som ligger nær slutten av en lang liste.
- Når vi skal fjerne elementer nær slutten.

Toveislister

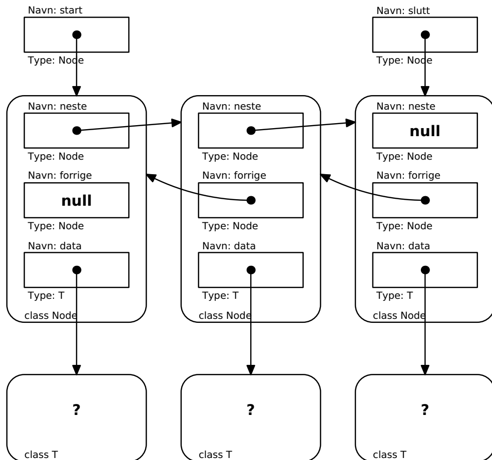
Derfor bruker noen toveislister («doubly-linked lists»):

```
class Node {
    Node neste = null;
    Node forrige = null;
    T data;

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



Da slipper vi å lete for å fjerne siste element:

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



Når skal jeg velge toveislister?

1. Når jeg ofte skal finne et av de siste elementene i en lang liste.
2. Når jeg ofte skal fjerne andre elementer enn det første i listen.

Ellers er det greit å holde seg til enveislister.

En mangel ved vår (og Javas) List

Listene våre (og Javas) kan kun inneholde objekter og ikke primitive verdier som int, char, boolean etc.

Hva kan vi gjøre med det?

Hva om vi lager en hjelpeklasse:

Heltall.java

```
public class Heltall {
    private int verdi;

    public Heltall(int i) {
        verdi = i;
    }

    public int intValue() {
        return verdi;
    }
```

... og en løsning

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

```
Liste<Heltall> lx = new Lenkeliste<>();
```

og vi kan bruke listen som normalt:

```
lx.add(new Heltall(12));

int v = lx.get(2).intValue();
```

Javas klasser for innpakking

Type	Klasse
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Disse klassene inneholder bl a:

***xxx*Value()** pakker ut en boks.

valueOf(*xxx* v) pakker v inn i en boks.

Nytt i Java:

Bruk heller metoden *Xxx*.valueOf(v) enn konstruktøren new *Xxx*(*xxx* v).

equals(*Xxx* p) sjekker likhet.

toString() omformer til en tekst.

parse*xxx*(String s) omformer fra en tekst.

Automatisk innkapsling

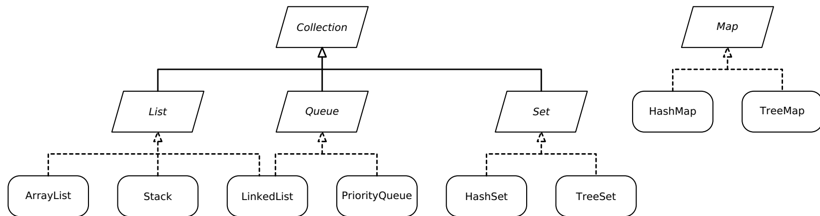
Innkapsling fungerer bra, men har en litt kronglete notasjon:

```
Liste<Integer> lx = new Lenkeliste<>();

lx.add(Integer.valueOf(12));
int v = lx.get(2).intValue();
```

Derfor har nyere versjoner av Java innført litt *syntaktisk sukker* og tillater automatisk inn- og utkapsling for standardklassene:

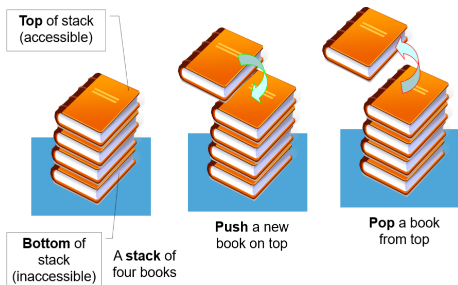
```
ArrayList<Integer> lx = new ArrayList<>();

lx.add(12);
int v = lx.get(2);
```

Javas Collection-klasser



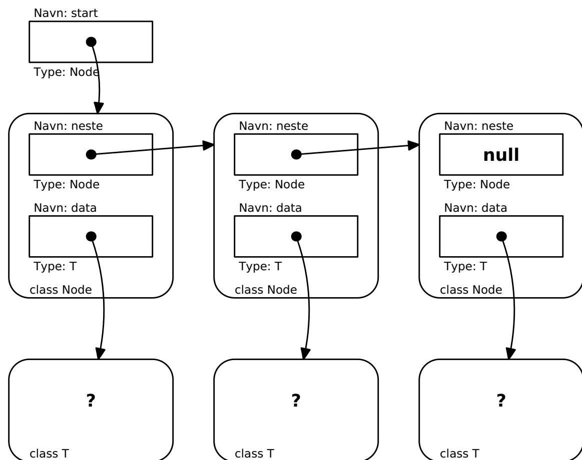
LIFO = Last in, first out

Stabler («stacks»)

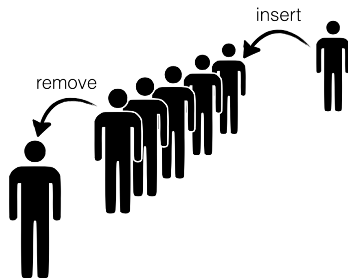


En spesiell form for lister er **stabler** der vi kun legger nye elementer på toppen («push-er») og kun henter fra samme ende («pop-er»).

LIFO = Last in, first out

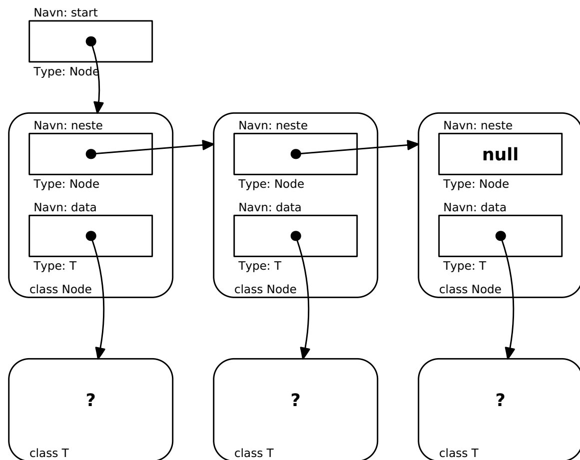


Køer («queues»)

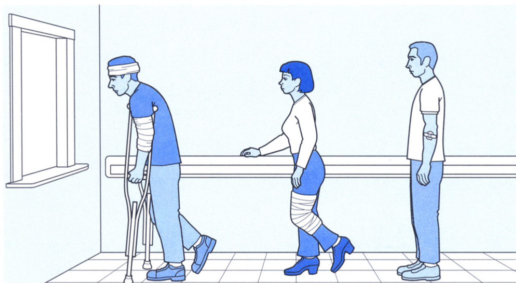


Køer er en annen spesiell form for lister der vi alltid setter inn nye elementer bakerst og henter dem ut forrerst.

FIFO = First in, first out



Prioritetskøer («Priority queues»)



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

Hvordan programmere en prioritetskø?

En prioritetskø må alltid holdes sortert. Nye elementer må derfor 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:

Comparable.java

```
public interface Comparable {
    public abstract int compareTo(Object 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

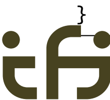
= 0 hvis vårt objekt er *likt* med det andre

> 0 hvis vårt objekt er *større* enn det andre

Et eksempel

Medaljeoversikt.java

```
public class Medaljeoversikt implements Comparable {
    protected String navn;
    protected int antGull, antSoelv, antBronse;

    Medaljeoversikt(String id, int g, int s, int b) {
        navn = id;  antGull = g;  antSoelv = s;  antBronse = b;
    }

    public int compareTo(Object annet) {
        Medaljeoversikt m = (Medaljeoversikt)annet;
        if (antGull < m.antGull) return -1;
        if (antGull > m.antGull) return 1;
        if (antSoelv < m.antSoelv) return -1;
        if (antSoelv > m.antSoelv) return 1;
        if (antBronse < m.antBronse) return -1;
        if (antBronse > m.antBronse) return 1;
        return 0;
    }
}
```



Et testprogram

TestMedaljer.java

```
class TestMedaljer {
    public static void main(String[] args) {
        Medaljeoversikt
            danmark = new Medaljeoversikt("Danmark", 0, 0, 0),
            finland = new Medaljeoversikt("Finland", 1, 1, 4),
            island = new Medaljeoversikt("Island", 0, 0, 0),
            norge = new Medaljeoversikt("Norge", 14, 14, 11),
            sverige = new Medaljeoversikt("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
```

Et testprogram til

Når en klasse implementerer Comparable, vet Java nok til å kunne sortere den effektivt:

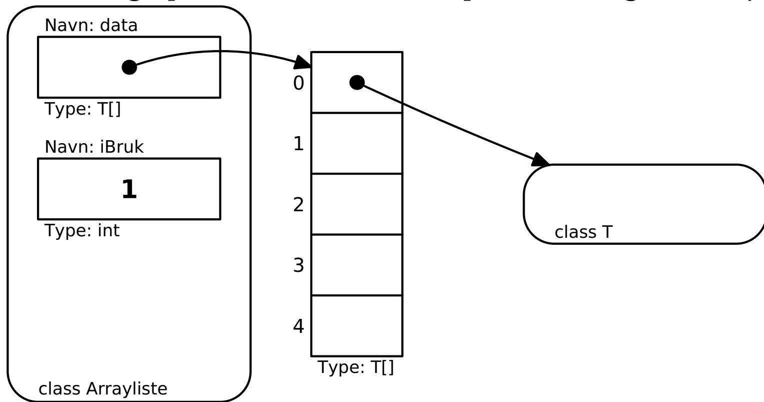
TestMedaljer2.java

```
import java.util.Arrays;

class TestMedaljer2 {
    public static void main(String[] args) {
        Medaljeoversikt[] land = {
            new Medaljeoversikt("Danmark", 0, 0, 0),
            new Medaljeoversikt("Finland", 1, 1, 4),
            new Medaljeoversikt("Island", 0, 0, 0),
            new Medaljeoversikt("Norge", 14, 14, 11),
            new Medaljeoversikt("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 en prioritetskø

Vi kan lage prioritetskø som en spesialisering av Arrayliste.



Rammen for en prioritetskø basert på Arrayliste kan se slik ut:

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

Kommentarer

- Klassen ArrayPrioKoe er en subklasse av Arrayliste.
- Klasseparameteren T *må* være en implementasjon av Comparable så vi får en sammenligning å sortere etter.
- Vi redefinerer kun metoden add siden innsettingen skal gjøres sortert.
- De øvrige metodene i Arratliste kan brukes som de er.

Når vi skal programmere add, må vi tenke på alle mulige situasjoner:

1. Listen kan være tom fra før av.
2. Hvis ikke, må vi lete oss frem til første element i listen som er større enn det vi skal sette inn.
    - Da må vi flytte det elementet og alle de resterende ett hakk opp.
    - Så kan vi sette inn det nye elementet på rett plass.
3. Hvis alle elementene i listen er mindre eller lik det nye elementet, må det nye elementet settes bakerst.

Hvordan programmere en 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 det nye elementet:
            super.add(x);
            return;
        }

        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
                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 og skal inn bakerst:
        super.add(x);
    }
}
```

Et testprogram

```
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 Berit");  ap.add("Irene");

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

```
Anne Berit
Dag
Ellen
Irene
Siri
Stein
```

Å gå gjennom en Liste

Vi kan gå gjennom en liste ved å hente elementene ett for ett:

```
Liste<String> lx = new Arrayliste<>();

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

En bedre notasjon

Det hadde vært fint å kunne skrive bare

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

og slippe å tenke på hva som er grensesnittet (size og get).

Javas interface Iterator

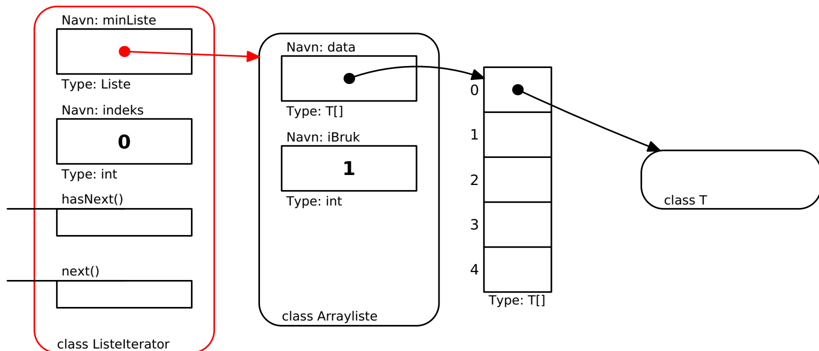
En slik elegant gjennomgang er mulig med en **iterator**:

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

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

Interface Iterator

For Arrayliste trenger vi en teller og en referanse til listen:



Interface Iterator

ListeIterator.java

```
import java.util.Iterator;

class ListeIterator<T> implements Iterator<T> {
    private Liste<T> minListe;
    private int indeks = 0;

    public ListeIterator(Liste<T> lx) {
        minListe = lx;
    }

    public boolean hasNext() {
        return indeks < minListe.size();
    }

    public T next() {
        return minListe.get(indeks++);
    }
}
```

Interface-et Iterable forteller at det finnes en tilhørende Iterator:

Liste.java

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

Så må vi lage en metode iterator (med *liten* forbokstav) som skaffer oss et Iterator-objekt (med *stor* forbokstav).

Arrayliste.java

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

    public ListeIterator iterator() {
        return new ListeIterator(this);
    }
```