IN1010

Repetisjon Uke 7
Beholdere, del 2

(Basert på Lars Tveitos presentasjon)

7. mars, 2025

Generiske beholdere

Fra forrige uke så hadde vi beholdere av en bestemt type.

Men som oftest ønsker vi at beholderene ikke er avhengig av det.
For å få til det så har Java det som kalles for generisk type
(generics) hvor vi bruker type-parametere.

Generiske klasser

```
class Skuff {
  Klesplagg plagg;
  Skuff(Klesplagg plagg) {
    this.plagg = plagg;
  }
}

Klesplagg bukse = new Klesplagg();
Skuff skuff = new Skuff(bukse);
```

```
class Skuff {
  Verktøy verktøy;
  Skuff(Verktøy verktøy) {
    this.verktøy = verktøy;
  }
}

Verktøy hammer = new Verktøy();
Skuff skuff = new Skuff(hammer);
```

Vi kan lage klasser som bruker objekter av forhåndsbestemte klasser,

men noen er så generelle at vi kan bruke de i andre sammenhenger

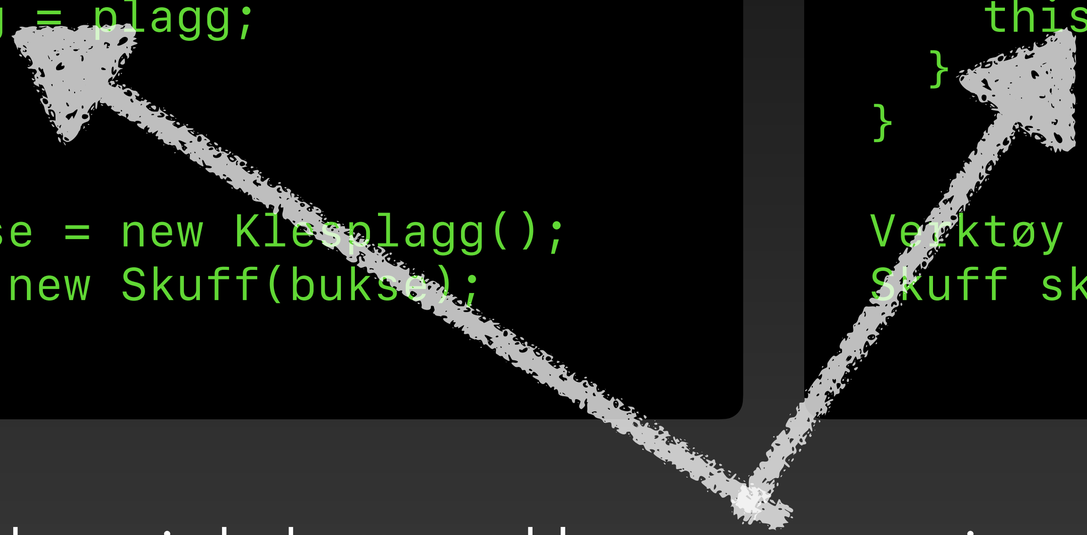
Generiske klasser

```
class Skuff {
  Klesplagg plagg;
  Skuff(Klesplagg plagg) {
    this.plagg = plagg;
  }
}

Klesplagg bukse = new Klesplagg();
Skuff skuff = new Skuff(bukse);
```

```
class Skuff {
  Verktøy verktøy;
  Skuff(Verktøy verktøy) {
    this.verktøy = verktøy;
  }
}

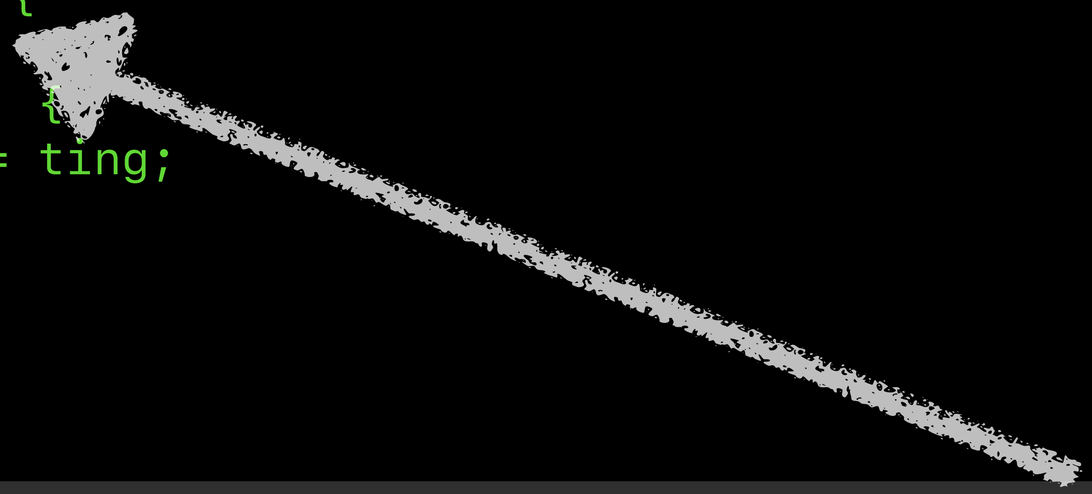
Verktøy hammer = new Verktøy();
Skuff skuff = new Skuff(hammer);
```



I Java kan vi da bytte ut klassene som vi ønsker å gjøre generiske

Generiske klasser

```
class Skuff<T> {
  T ting;
  Skuff(T ting) {
    this.ting = ting;
  }
}
```



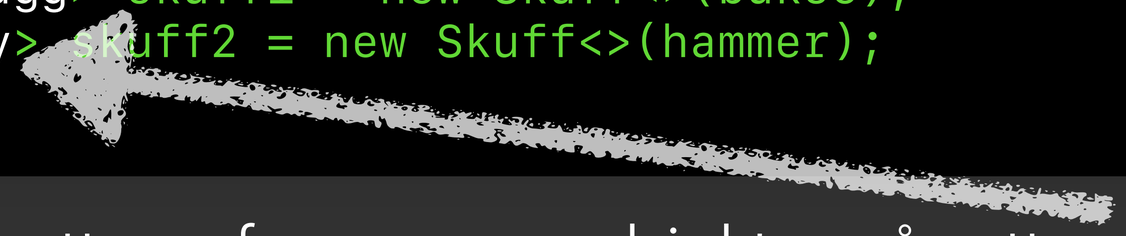
For å vise at vi vil ha en generisk klasse bruker vi <T> etter klassenavnet

T viser hva vi vil at Java skal bytte ut med klassen som vi vil bruke

Generiske klasser

```
class Skuff<T> {
  T ting;
  Skuff(T ting) {
    this.ting = ting;
  }
}

Skuff<Klesplagg> skuff1 = new Skuff<>(bukse);
Skuff<Verktøy> skuff2 = new Skuff<>(hammer);
```



Når vi oppretter referanser og objekter, så setter vi inn klassene vi vil erstatte

Merk at vi ikke trenger å oppgi typen i "new"

Generiske klasser

```
class Skuff<T> {
  T ting;
  Skuff(T ting) {
    this.ting = ting;
  }
}

Skuff<Klesplagg> skuff1 = new Skuff<>(bukse);
Skuff<Verktøy> skuff2 = new Skuff<>(hammer);
```

Vi må ikke bruke "T", vi kan kalle det for hva som helst, men mest vanlig er å bruke bare en bokstav. Så lenge det står mellom < og > så blir det erstattet ved kompilering

Generiske klasser

```
class Skuff<T> {
  T ting;
  Skuff(T ting) {
    this.ting = ting;
  }
}

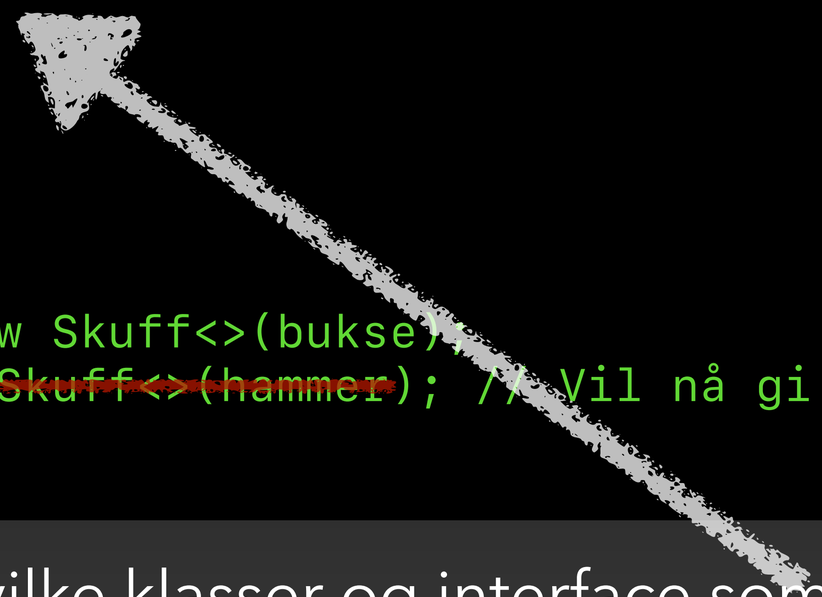
Skuff<Klesplagg> skuff1 = new Skuff<>(bukse);
Skuff<Verktøy> skuff2 = new Skuff<>(hammer);
```

De vanligste er E (element), K (key), N (number), T (type), V (value)

Generiske klasser

```
class Skuff<T extends Klesplagg> {
  T ting;
  Skuff(T ting) {
    this.ting = ting;
  }
}

Skuff<Klesplagg> skuff1 = new Skuff<>(bukse);
Skuff<Verktøy> skuff2 = new Skuff<>(hammer); // Vil nå gi feilmelding
```



Det er mulig å begrense hvilke klasser og interface som kan brukes til å erstatte

Men pass på at vi bruker *extends* både for klasser og interface!

Generiske klasser

Begrensing

```
class Skuff<T extends Klesplagg> {
  T ting;
  Skuff(T ting) {
    this.ting = ting;
  }
}
```

Arv

```
class Skuff<T> {
  T ting;
  Skuff(T ting) {
    this.ting = ting;
  }
}

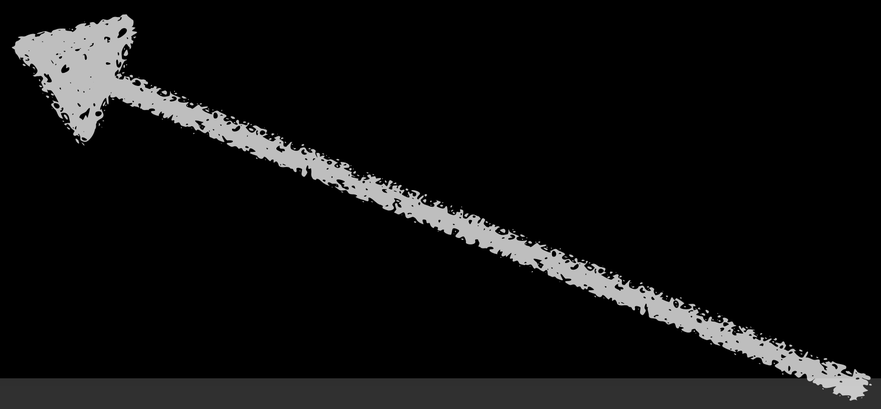
class StorSkuff<T> extends Skuff<T> {
}
```

Vi kan arve på samme måte som vanlige klasser.

Pass på forskjellen mellom arv og begrensing.

Generiske klasser

```
class Ordbok<K, V> {
  /* … */
}
```



Det er mulig å bruke flere generiske typer i en klasse,

som her hvor vi har key og value som type-paramentere i en ordbok.

Generiske typer

Det er i beholdere og klasser som har lignende egenskaper som vi som oftest bruker generiske typer.

Det er også vanlig å bruke for å unngå typekonverteringer (casting)

Generiske typer vil ikke ha en "type" som vi kan bruke f.eks. *instanceof* på

Generiske metoder

```
class Hjelpemetoder {
  public static <T> List<T> konverterTilArrayList(T[] array) {
  List<T> liste = new ArrayList<>();
  for (T element : array) {
    liste.add(element);
  }
  return liste;
}

String[] varer = {"Melk", "Brød", "Ost", "Kake"};
List<String> vareListe = Hjelpemetoder.konverterTilArrayList(varer);
```

Vi kan også bruke generiske typer i metoder, selv når klassen ikke er generisk

Men det kan fort bli uoversiktlig

Itererbare klasser (Iterator)

Itererbare klasser betyr at de kan "løpe" gjennom en serie med objekter

Beholdere vil vi som oftest at skal ha den egenskapen, siden vi kan bruke de i bl.a. for-løkker

I Java så implementeres de ved å bruke interfacet *Iterable<T>* og *Iterator<T>*

Itererbare klasser

```
class PrioKø<T extends Comparable<T>> implements Iterable<T> {
    @Override
    public Iterator<T> iterator() {
        return new KøIterator();
    }

    private class KøIterator implements Iterator<T> {
        /* … */
    }
}
```

Som en del av itererbare klasser så må vi implementere *Iterable<T>*

Den har en metode *iterator()* som returnerer en type *Iterator<T>*

Itererbare klasser

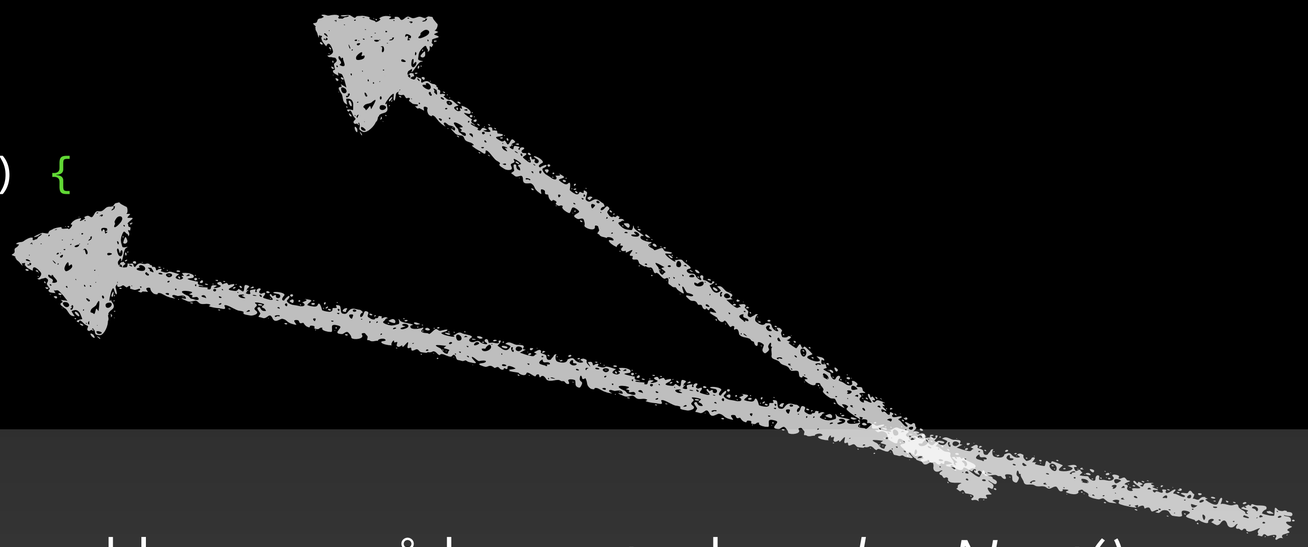
```
class PrioKø<T extends Comparable<T>> implements Iterable<T> {
    @Override
    public Iterator<T> iterator() {
        return new KøIterator();
    }

    private class KøIterator implements Iterator<T> {
        /* … */
    }
}
```

Iterator<T> er det mest vanlig at vi implementerer som en del av en intern klasse inne i klassen vår

Itererbare klasser

```
private class KøIterator implements Iterator<T> {
    private Node<T> gjeldende = første;
    @Override
    public boolean hasNext() {
        /* … */
    }
    @Override
    public T next() {
        /* … */
    }
}
```



Denne klassen må ha metodene *hasNext()* og *next()*

Itererbare klasser

```
private class KøIterator implements Iterator<T> {
    private Node<T> gjeldende = første;
    @Override
    public boolean hasNext() {
        /* … */
    }
    @Override
    public T next() {
        /* … */
    }
}
```

Metoden *hasNext()* sjekker om vi har et nytt element i iterasjonskøen

Itererbare klasser

```
@Override
public boolean hasNext() {
    return gjeldende != null;
}
```

I eksempelet som er en lenket liste, så sjekker vi at gjeldende node ikke er *null*

Itererbare klasser

```
private class KøIterator implements Iterator<T> {
    private Node<T> gjeldende = første;
    @Override
    public boolean hasNext() {
        /* … */
    }
    @Override
    public T next() {
        /* … */
    }
}
```

Så vil metoden next() gi ut det neste elementet hvis det eksisterer

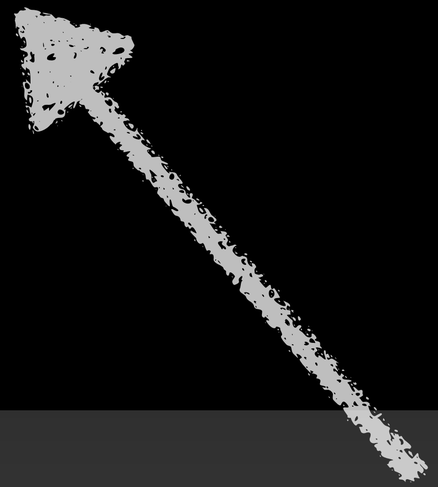
Itererbare klasser

```
@Override
public T next() {
    if (!hasNext()) {
        throw new NoSuchElementException();
    }
    T element = gjeldende.element;
    gjeldende = gjeldende.neste;
    return element;
}
```

Implementasjon i lenkelisten

Itererbare klasser

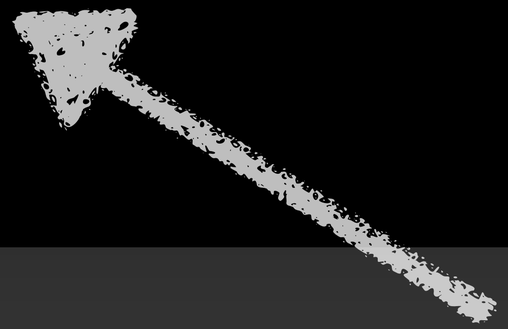
```
@Override
public T next() {
    if (!hasNext()) {
        throw new NoSuchElementException();
    }
    T element = gjeldende.element;
    gjeldende = gjeldende.neste;
    return element;
}
```



Først sjekker vi at det faktisk eksisterer, og kaster et unntak om det ikke gjør det

Itererbare klasser

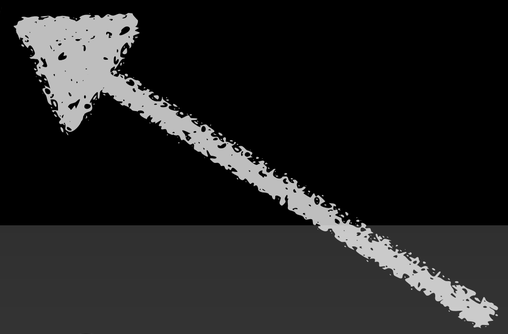
```
@Override
public T next() {
    if (!hasNext()) {
        throw new NoSuchElementException();
    }
    T element = gjeldende.element;
    gjeldende = gjeldende.neste;
    return element;
}
```



Vi gjør klar slik at vi peker på neste element

Itererbare klasser

```
@Override
public T next() {
    if (!hasNext()) {
        throw new NoSuchElementException();
    }
    T element = gjeldende.element;
    gjeldende = gjeldende.neste;
    return element;
}
```



Og returnerer det gjeldende elementet

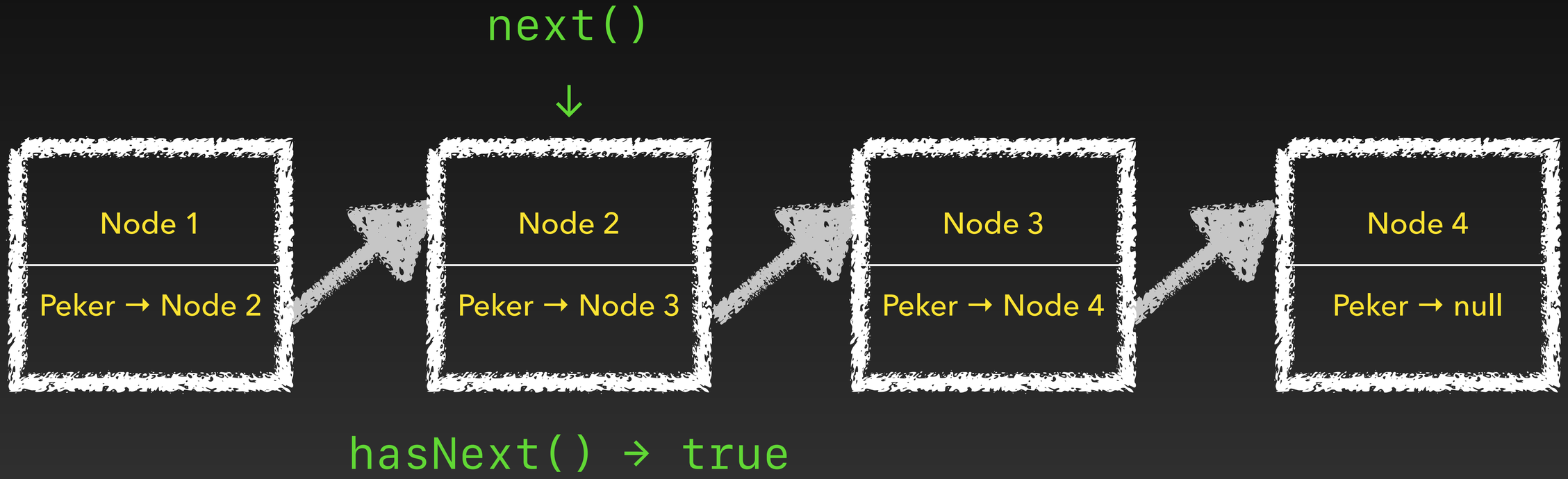
Itererbare klasser



`hasNext() → true`

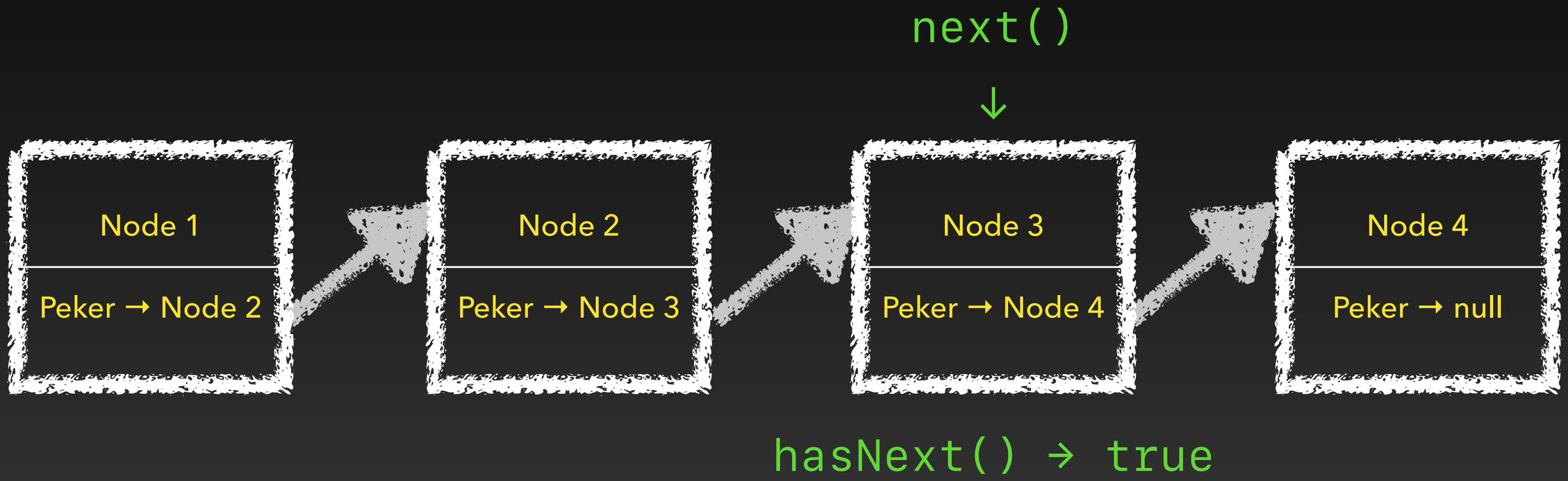
```
for (Dyr dyr : dyreListe) { /* … */ }
```

Itererbare klasser




```
for (Dyr dyr : dyreListe) { /* … */ }
```

Itererbare klasser




```
for (Dyr dyr : dyreListe) { /* … */ }
```

Itererbare klasser




```
for (Dyr dyr : dyreListe) { /* … */ }
```

Comparable

Interface for å sammenligne objekter av en type

Brukes også for å sortere

Har en metode compareTo() som returnerer negative tall, 0 eller positive tall for å markere om noe er mindre, lik, eller større

```
public interface Comparable<T>
```

Livekoding

Dyreklinikk

Neste uke

Midtveiseksamen

Spesielle temaer