



NTNU

Kunnskap for en bedre verden

TDT4100 - Objektorientert programmering

Iterator-teknikken

Børge Haugset

Dag Olav Kjellemo

Sist uke: Comparable/Comparator

- **Interface**: En mal for en kontrakt
 - Fra utsiden definerer det hva vi har behov for
 - Fra innsiden beskriver det metodene vi må implementere
- **Comparable**: Interface for Klasse som sier hvordan objektene (instanser av Klassen) skal sorteres
- **Comparator**: Egen hjelpeklasse som forteller hvordan man skal sortere elementer av *Klasse*
- Ved å implementere disse støtter vi også *Collections.sort()* og *liste.sort()*

Vi separerer hvordan en sammenligner fra hvordan sorteringen gjøres .

Når definerer vi Comparable?

- Hvorfor kan man kjøre
 - For kan man kalle *Collections.sort(listeMedStrenger)* selv om man ikke lager noen comparable/comparator?
 - Selvsagt fordi String implementerer interfacet *Comparable*!
- *Comparable* er mest egnet for klasser hvor objektene har en naturlig/vanlig ordning.
- *Comparator* er mer fleksibel, og vi skal I dag se hvordan vi enkel kan definer slike.

Arv og Grensesnitt

- Klasser kan implementere flere grensesnitt, men bare arve fra en foreldre-klasse (superklasse).
- Et interface har ikke tilstand, og implementerer ikke metoder (**men mer om dette senere**)
- Ved arv får man alt fra superklassen, og en får en tett kobling mellom super- og sub-klasser, på godt og vondt!
- Begge gir mulighet for polymorfisme.

Læringsmål for forelesningen

- Iterator-teknikken
 - Hva er en Iterator og hvorfor bruke den?
 - Hvordan virker en Iterator?
 - Iterable-grensesnittet og for-løkker
- Eksempel med bruk av Iterator og Iterable-grensesnittene
 - Vi lager en Iterator for tegnene i en String
 - En Person kan ha flere Person som barn, og vi kan iterere over dem
- Hvorfor fungerer *for String streng : strengListe* ?

java.util.Iterator/Iterable

Standard Java-grensesnitt for iterasjon

Iterasjon (over tegn) – hva er felles?

```
List<Character> charList;  
...  
for (int i = 0; i < charList.size(); i++) {  
    Character c = charList.get(i);  
    System.out.println(c);  
}
```

```
String s;  
...  
for (int i = 0; i < s.length(); i++) {  
    Character c = s.charAt(i);  
    System.out.println(c);  
}
```

```
char[] charArray;  
...  
for (int i = 0; i < charArray.length; i++) {  
    Character c = charArray[i];  
    System.out.println(c);  
}
```

java.util.Iterator – hva og hvorfor

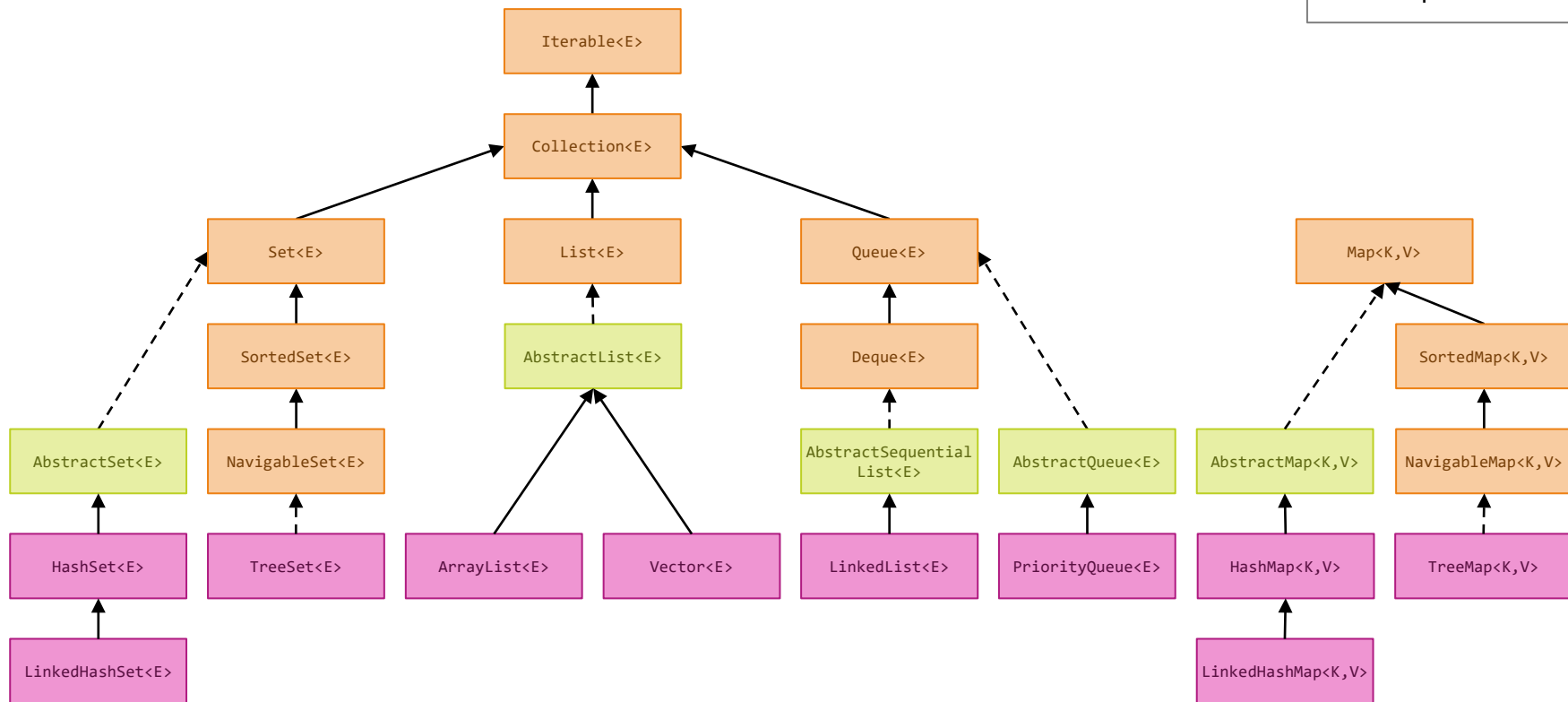
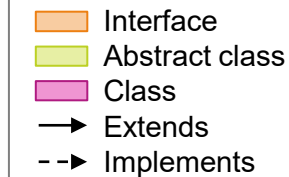
- Å bruke ulike typer samlinger av objekter, som lister (List), stabler (Stack) og sett (Set) krever ofte at en *itererer* (går gjennom) alle elementene i samlingen
- Iterasjon med List:

```
List objekter = ...  
for (int i = 0; i < objekter.size(); i++) {  
    Object o = objekter.get(i);  
    System.out.println("Neste: " + o);  
}
```
- Eksempel på *indeksbasert* iterasjon
 - int-variabel brukes som løkkevariabel
 - bruker løkkevariabel for å hente ut objekt

Men hva gjør en i det generelle tilfellet?

Iterator-teknikken/grensesnitt gir en uniform måte å iterere over elementer

Java collections hierarki



java.util.Iterator

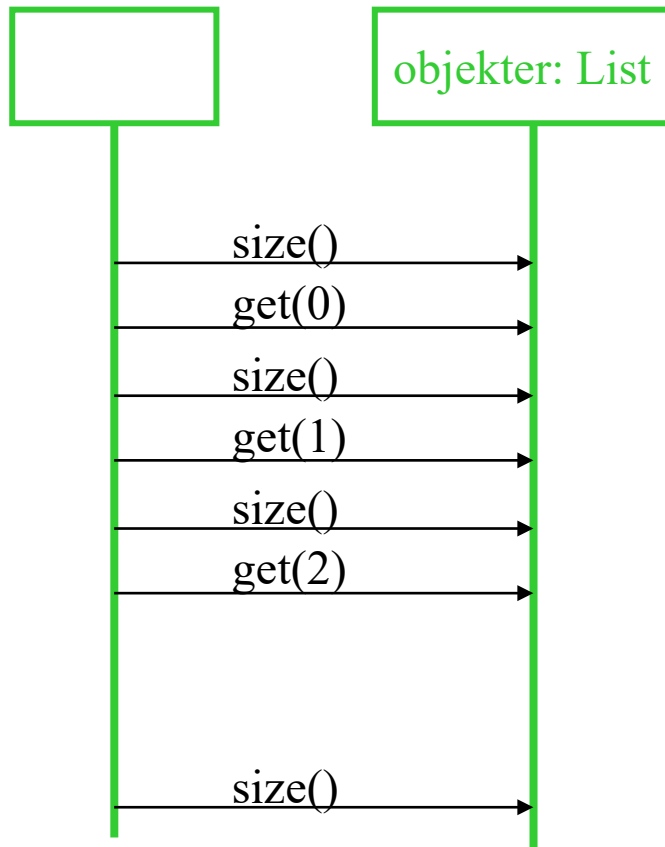
- En såkalt *iterator* er et objekt som "genererer" objekter
 - iterator.**hasNext()** sier om det er flere objekter å generere
 - iterator.**next()** genererer neste objekt (som "brukes opp")
- Kode-eksempel:

```
List<Person> personer = ...
Iterator<Person> it = personer.iterator();
while (it.hasNext()) {
    Person p = it.next();
    System.out.println("Neste: " + p);
}
```

java.util.Iterator

- Iteratoren "husker" hvor langt i samlingen en er kommet
 - **hasNext()** returnerer true så lenge vi ikke har kommet gjennom hele lista
 - **next()** vil hver gang returnere neste element og flytte seg et trinn utover i lista
- **iterator()**-metoden er definert for alle typer samlinger (Collection)
 - alle List- og Set-implementasjoner har den

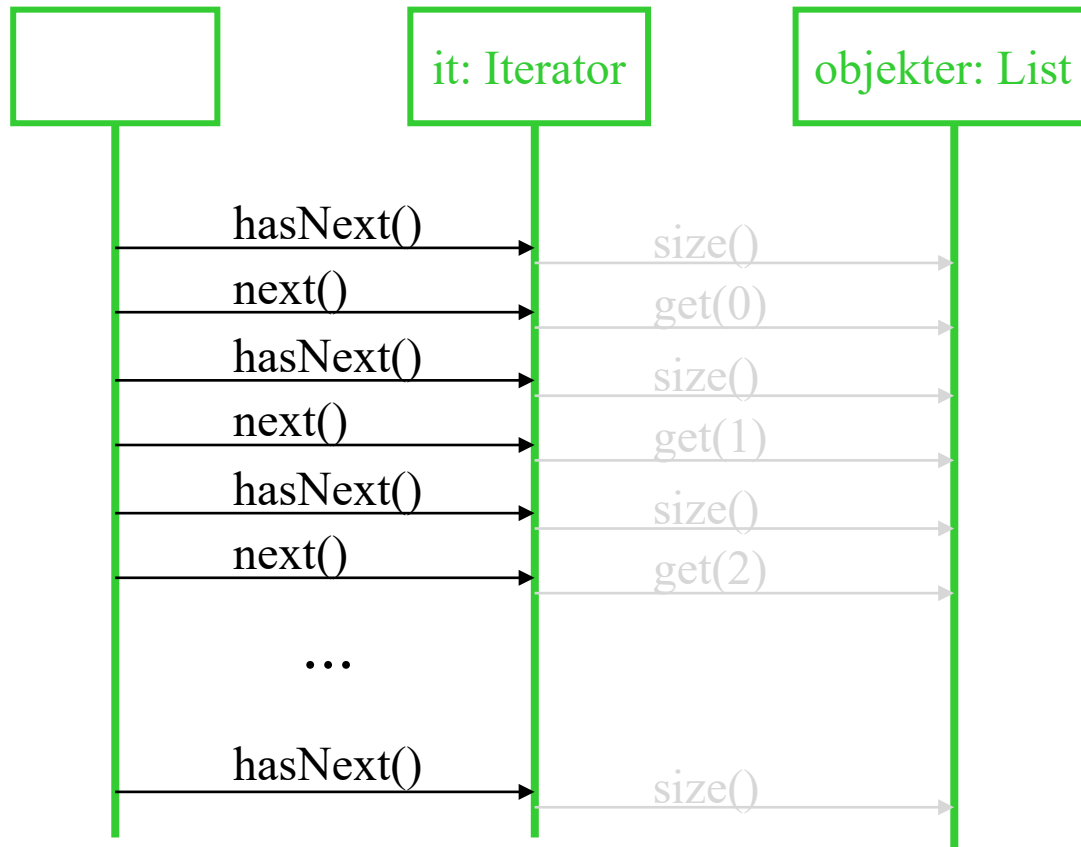
Sekvensdiagram for iterasjon med List





NTNU

Sekvensdiagram for iterasjon med Iterator



Et Iterator-objekt husker hvor langt vi er kommet, og bruker den underliggende lista til å

- sjekke om vi er ferdig og**
- hente ut neste element**

Hvorfor introdusere en slik teknikk, den er jo ikke noe lettere å bruke?

Et par eksempler

- RandomEvenNumber – implementerer *Iterator*
- Streng – implementerer *Iterable*
- Mergeliterator – bruke to iteratorer

Bruk av iterator

- *Uniform* måte å iterere over en samling objekter
 - Uavhengig av om det er en posisjonsbasert List-implementasjon eller en annen type samling
 - Uavhengig av metode som benyttes for å hente ut objekter (tabell[i] eller list.get(i))
- Gir mindre avhengighet mellom kode som aksesserer en samling og kode som implementerer en samling
- Kan f.eks. lage kode for å summere tall, uten å vite hvor tallene kommer fra (som en netttjeneste!)



NTNU

Iterasjon over personer-liste (List<Person> personer)

Kode for å finne et bestemt element:

```
List<Person> personer = ...  
// Vi skal se etter personer med navn  
// "Jan Johansen"  
for (int i = 0; i < personer.size(); i++) {  
    Person p = personer.get(i);  
    if ("Jan Johansen".equals(p.getName())) {  
        // send e-post til Christine Koht  
        ...  
    }  
}
```

Iterasjon over tabell

Den vanligste måten å iterere er vha. av for og en løkkevariabel av typen int:

```
// Kode for å finne et bestemt element:  
Person[] personer = ...  
// Vi skal se etter personer med navn  
// "Jan Johansen"  
for (int i = 0; i < personer.length; i++) {  
    Person p = personer[i];  
    if ("Jan Johansen".equals(p.getName())) {  
        // send e-post til Christine Koht  
        ...  
    }  
}
```

Standard løkkestruktur

- Løkka inneholder typisk kode som:
 - håndterer løkkevariabelen (for-løkker skiller tydelig mellom disse tre elementene, mens while-løkker har dem mer implisitt)
 - deklarerer løkkevariabelen og gir den en initialverdi: `int i = 0`
 - sjekker om løkka skal gjentas: `i < personer.length` e.l.
 - beregner neste verdi for løkkevariabelen (`i++` / `i+=1` / `i=i+1`)
 - håndterer objektet for hver runde i løkka
 - identifiserer hvilket objekt som skal behandles:
`Person p = personer[i]`
 - bruker evt. manipulerer objektet
`if ("Jan Johansen".equals(p.getName())) { ... }`

Standard løkkestruktur

- Viktig observasjoner:
 - De først fire punktene (init, test, steg, objekt) er avhengig av hva vi itererer over
 - **Kun det siste punktet (løkke kropp) er avhengig av hva vi ønsker å gjøre**
 - *En iterator innkapsler de fire første punktene, slik at løkka blir uavhengig av hva vi itererer over.*
 - Kompleksiteten i løkka dyttes inn i iterator-objektet og dermed over på iterator-koderen.
 - Alle datastrukturer burde ha en tilhørende iterator-klasse, som f.eks. ArrayList har, og en iterator()-metode.

Iterasjon over personer-liste med Iterator<Person>

Kode for å finne et bestemt element:

```
List<Person> personer = ...  
// Vi skal se etter personer med navn  
// "Jan Johansen"  
Iterator<Person> it = personer.iterator();  
while (it.hasNext()) {  
    Person p = it.next();  
    if ("Jan Johansen".equals(p.getName())) {  
        // send e-post til Christine Koht  
        ...  
    }  
}
```

List har en iterator, men hva med String og tabeller?

```
List<Character> charList;  
...  
for (int i = 0; i < charList.size(); i++) {  
    Character c = charList.get(i);  
    print(c);  
}
```

```
String s;  
...  
for (int i = 0; i < s.length(); i++) {  
    Character c = s.charAt(i);  
    print(c);  
}
```

```
char[] charArray;  
...  
for (int i = 0; i < charArray.length; i++) {  
    Character c = charArray[i];  
    print(c);  
}
```

*String er ikke iterable,
man vi kan lage våre
egne iteratorer!*

Iterable (1)

- Iterable er et grensesnitt som kun tilbyr én metode: `iterator()`
- Enkelt sagt så implementeres det av klasser som tilbyr noe å iterere over.
- En kan da bruke en spesiell for-variant for iterasjon:

```
List<Person> personer = ...  
for (Person person: personer) {  
    ...  
}
```

- Samme som

```
List<Person> personer = ...  
Iterator<Person> it = personer.iterator();  
while (it.hasNext()) {  
    Person person = it.next();  
    ...  
}
```

Iterable (2)

Generell form (der X kan være noe helt annet enn i Collections)

```
Iterable<X> xer = ...  
for(X x : xer) {  
    ...  
}
```

Samme som

```
Iterable<X> xer = ...  
Iterator<X> it = xer.iterator();  
while(it.hasNext()) {  
    X x = it.next();  
    ...  
}
```

Iterable (3)

Ved å implementere Iterable i egne klasser, så kan en bruke denne for-varianten til iterasjon

```
class Person implements Iterable<Person> {  
    private List<Person> children;  
  
    public Iterator<Person> iterator() {  
        return children.iterator();  
    }  
}
```

Kan da bruke:

```
Person father = ...  
for (Person child: father) {  
    // kode som bruker child  
    ...  
}
```

Husker dere dictionaries fra Python?

- I Java har vi grensesnittet Map

Map *kan* itereres gjennom, (*HS.java*)

Neste gang tar vi dette videre

- Mer om funksjonelle grensesnitt (kun én metode)
- Anonyme, indre klasser
- Lambda-uttrykk