Algoritmer og datastrukturer
Kapittel 3 – Delkapittel 3.1

3.1 En beholder

3.1.1 En beholder

En pappeske er en beholder

En beholder er noe vi kan legge ting i. De fleste av oss får stadig papirer som det er nødvendig å ta vare på. Noen legger papirene i en haug på skrivebordet, noen legger dem i en skuff, andre legger dem i en eske og de mest ryddige setter papirene i en perm. Alle disse «beholderne» har som egenskap at de er «åpne», dvs. vi kan bla gjennom innholdet. Vi kan for eksempel se på ett og ett ark og dermed finne (og eventuelt fjerne) et hvilket som helst ark som måtte være lagt inn der. Antallet i «beholderen» kan vi for eksempel finne ved å bla gjennom og telle opp.

```
public interface Beholder<T> extends Iterable<T>  // foreløpig versjon
{
  public boolean leggInn(T verdi);   // legger inn i beholderen
  public Iterator<T> iterator();     // returnerer en iterator

  // andre aktuelle metoder
}
```

Programkode 3.1.1 a)

Metoden *leggInn*() i *Beholder* legger *verdi* av typen *T* i beholderen og returnerer true. Den returnerer false hvis duplikater ikke er tillatt og *verdi* finnes fra før. Metoden *iterator*() returnerer en Iterator, dvs. «noe» som vi kan «bla gjennom» beholderen med. Det er strengt tatt ikke nødvendig å ta den med siden den uansett arves fra grensesnittet Iterable:

```
public interface Iterable<T>                          // ligger i java.lang
{
  Iterator<T> iterator();                             // en iterator

  default void forEach(Consumer<? super T> action);   // traversering
  default  Spliterator spliterator();                 // parallellprosessering
}
```

Programkode 3.1.1 b)

En *iterator* er en instans av en klasse som implementerer grensesnittet Iterator:

```
public interface Iterator<T>                    // ligger i java.util
{
  public boolean hasNext();                     // er det flere igjen?
  public T next();                              // returnerer neste verdi

  default void remove();                        // fjerner sist returnerte verdi
  default void forEachRemaining(Consumer<? super T> action);   // traversering
}
```

Programkode 3.1.1 c)

En *iterator* kan ses på som en «innretning» som gjør det mulig å «bla» eller *iterere* gjennom en beholder. Å iterere betyr å utføre (gjenta) i rekkefølge. Itereringen skjer ved gjentatte kall på metoden *next*(). Da kommer hele innholdet fortløpende (en verdi om gangen) i en eller annen rekkefølge. Logisk sett burde det vært en metode *first*() som gir den første i rekkefølgen. Deretter skulle kall på *next*() gi den neste i forhold til det vi fikk sist. Men det er isteden laget slik, som kanskje er litt ulogisk, at første kall på *next*(), gir den første i rekkefølgen. Metoden *hasNext*() sjekker om det er flere igjen å se på i «beholderen».

En *iterator* brukes normalt kun til å «bla»/iterere gjennom «beholderen». Men i mange tilfeller er det også mulig å fjerne objekter (ved hjelp av metoden remove) mens vi «blar». Metoden er satt opp som en standard metode (default) i grensesnittet. Det betyr at den formelt sett er kodet, men som «unsupported» (den kaster et unntak). Det betyr at hvis vi ønsker at den skal kunne brukes til å fjerne verdier, må vi selv kode den. Den er satt opp slik i grensesnittet Iterator:

```
default void remove()
{
  throw new UnsupportedOperationException("remove");
}
```

Programkode* 3.1.1 *d)

Hvis vi skal kode *remove*(), er regelen at den skal fjerne det objektet som sist ble returnert av metoden *next*(). Det betyr spesielt at det er ulovlig å kalle *remove*() først, dvs. før det i det hele tatt er gjort et kall på *next*(). Videre skal det også være ulovlig å kalle remove() to ganger på rad (dvs. uten at det har vært et kall på *next*() i mellomtiden). I så fall skal det kastes en *IllegalStateException*.

En «traversering» gjennom en beholder kan gjøres ved gjentatte kall på *next*() – enten implisitt (forskjellige typer *forAlle*-løkker) eller eksplisitt (vanlige *for*- og *while*-løkker). La *beholder* være et beholderobjekt med T som typeparameter. Da kan en *forAlle*-løkke som f.eks. skriver ut hver verdi i beholderen, lages slik:

```
for (T verdi : beholder) System.out.print(verdi + " ");  // en forAlle-løkke
```

Programkode* 3.1.1 *e)

En *forAlle*-løkke har «usynlige» kall på *next*() og *hasNext*(). De utføres implisitt.

Vi kan også bruke beholderens *forEach*-metode eller iteratorens *forEachRemaining*-metode. Begge har en Consumer som parameter. Hvis *beholder* er et beholderobjekt med T som datatype, kan en utskrift gjøres slik (ved å bruke et lambda-uttrykk):

```
beholder.forEach(verdi -> System.out.print(verdi + " "));
beholder.iterator().forEachRemaining(verdi -> System.out.print(verdi + " "));
```

Programkode* 3.1.1 *f)

En vanlig *for*-løkke som skriver ut innholdet av *beholder*, kan f.eks. lages slik:

```
for (Iterator<T> i = beholder.iterator(); i.hasNext(); )  // for-løkke
{
  T verdi = i.next();               // henter neste i beholderen
  System.out.print(verdi + " ");    // skriver ut
}
```

Programkode* 3.1.1 *g)

Alternativt kan vi bruke en *while*-løkke:

```
Iterator<T> i = beholder.iterator();   // henter iteratoren

while (i.hasNext())                    // er det flere?
{
  System.out.print(i.next() + " ");    // skriver ut neste
}
```

Programkode* 3.1.1 *h)

Flg. unntak kommer ved feil bruk av *next*() og *remove*():

Operasjon	Unntak
Kall på *next* hvis ikke flere igjen (eller tom beholder)	NoSuchElementException
Kall på *remove* før første *next* eller to kall på rad	IllegalStateException
Kall på *remove* hvis den er «utilgjengelig»	UnsupportedOperationException

Tabell 3.1.1 : Unntak som kastes fra *next* og *remove* i en iterator

I grensesnittet Beholder ble det satt opp kun to metoder, men det er gjort plass til flere. Strengt tatt trenger vi ikke flere fordi alle aktuelle behov kan normalt dekkes av de to. Men det er praktisk (og mer effektivt) å ha noen flere metoder. Vi tar dem med i grensesnittet:

```
import java.util.*;
import java.util.function.Predicate;

public interface Beholder<T> extends Iterable<T>  // ny versjon
{
  public boolean leggInn(T verdi);    // legger inn verdi i beholderen
  public boolean inneholder(T verdi); // sjekker om den inneholder verdi
  public boolean fjern(T verdi);      // fjerner verdi fra beholderen
  public int antall();                // returnerer antallet i beholderen
  public boolean tom();               // sjekker om beholderen er tom
  public void nullstill();            // tømmer beholderen
  public Iterator<T> iterator();      // returnerer en iterator

  default boolean fjernHvis(Predicate<? super T> p)  // betingelsesfjerning
  {
    Objects.requireNonNull(p);                       // kaster unntak

    boolean fjernet = false;
    for (Iterator<T> i = iterator(); i.hasNext(); )  // løkke
    {
      if (p.test(i.next()))                          // betingelsen
      {
        i.remove(); fjernet = true;                  // fjerner
      }
    }
    return fjernet;
  }
} // grensesnitt Beholder
```

Programkode* 3.1.1 *i)

Legg spesielt merke til metoden *fjernHvis*(). Den er standard (default). Slike metoder må kodes i grensesnittet og alle klasser som implementerer grensesnittet vil arve dem. Metoden har et predikat som parameter (se Avsnitt 1.10.5). Den benytter metoden *remove*() i iteratoren. Det betyr at hvis den ikke er kodet (dvs. hvis den kun kaster et unntak), så vil ikke *fjernHvis*() virke.

1) En klasse som implementerer grensesnittet *Beholder*, kan lages på to måter. Det kan være en konkret klasse med en intern datastruktur der metodene kodes mest mulig effektivt med hensyn på den gitte strukturen. Det ser vi på i delkapitlene 3.2 og 3.3.

2) Alternativt kan vi lage en mellomløsning ved først å lage en abstrakt klasse der metodene leggInn(T verdi) og iterator() gjøres abstrakte og alle de andre metodene kodes ved hjelp av dem. Da må imidlertid metoden *remove* i iteratoren kodes. Det betyr at når vi skal lage en konkret subklasse av den, holder det å kode de to abstrakte metodene. Det vil kunne bli ineffektivt for flere av metodene, men det vil virke. Eventuelt kan vi omkode (overskrive) de minst effektive metodene. Vi ser mer på denne idéen i neste avsnitt.

I biblioteket java.util finnes det et grensesnitt omtrent som vår beholder. Det heter Collection og inneholder blant andre disse metodene:

```
public interface Collection<E> extends Iterable<E>
{
  boolean add(E e);
  boolean contains(Object o);
  boolean remove(Object o);
  default boolean removeIf(Predicate<? super E> filter);
  int size();
  boolean isEmpty();
  void clear();
  Iterator<E> iterator();

  // + noen andre metoder
}
```

Programkode* 3.1.1 *j)

Dette grensesnittet danner basis for en hel serie klasser i java.util. F.eks. klassene ArrayList, LinkedList, TreeSet og HashSet. Det finnes også en AbstractCollection. Det står *Object o* istedenfor *E e* i metodene *contains* og *remove* pga. bakoverkompabiliteten. Grensesnittet *Collection* fantes før den generiske teknikken kom.

Oppgaver til Avsnitt 3.1.1

1. Legg Beholder fra *Programkode* 3.1.1 *i)* inn under hjelpeklasser i ditt prosjekt.
2. En *forAlle*-løkke kan iterere i en vanlig tabell og i en instans av en klasse som implementerer *Iterable*. Lag kode med en heltallstabell der noen tall er mindre enn og noen større enn 10. Bruk så en *forAlle*-løkke til å finne antallet som er større enn 10.

3.1.2 En abstrakt beholder

En *AbstraktBeholder* er en klasse som implementerer grensesnittet *Beholder*. De to metodene *leggInn* og *iterator* gjøres abstrakte, mens alle de andre metodene kodes eksplisitt eller implisitt ved hjelp av dem. I flg. klasse har alle ikke-abstrakte metoder (unntatt `tom()` som er ferdigkodet) blitt satt opp med «tom» (foreløpig) kode:

```
public abstract class AbstraktBeholder<T> implements Beholder<T>
{
  public abstract boolean leggInn(T t);    // en abstrakt metode

  public boolean inneholder(T t)
  {
    return false;   // foreløpig kode
  }

  public boolean fjern(T t)
  {
    return false;   // foreløpig kode
  }

  public int antall()
  {
    return 0;   // foreløpig kode
  }

  public boolean tom()
  {
    return antall() == 0;   // ferdig kode
  }

  public void nullstill()
  {
     // foreløpig kode
  }

  public abstract Iterator<T> iterator();    // en abstrakt metode

  public String toString()
  {
    return null;   // foreløpig kode
  }
}
```

Programkode* 3.1.2 *a)

Metoden `tom()` er kodet ved hjelp av metoden `antall()` og det vil virke siden en beholder er tom hvis og bare hvis antallet objekter er 0. Metoden `antall()` kan igjen kodes ved hjelp av iteratoren. Vi kan f.eks. la opptellingen foregå i en *forAlle*-løkke:

```
public int antall()
{
  int antall = 0;
  for (T t : this) antall++;   // bruker en forAlle-løkke
  return antall;
}
```

Programkode* 3.1.2 *b)

Metoden `inneholder(T t)` kan også kodes ved hjelp av en *forAlle*-løkke. Da tester vi hvert objekt (metoden `equals`) som løkken gir oss og returnerer *true* hvis vi finner objektet *t*. Hvis løkken avslutter uten at vi finner *t*, returneres *false*. Se *Oppgave* 1. Metodene `fjern(T t)` må benytte mulighetene i iteratoren. Hvis beholderen tillater at det legges inn null-verdier, må vi passe nøye på når en innlagt null-verdi etterpå skal fjernes:

```
public boolean fjern(T t)
{
  Iterator<T> i = iterator();

  if (t == null)                // fjerner en eventuell null-verdi
  {
    while (i.hasNext())         // flere igjen
    {
      if (i.next() == null)     // sammenligner
      {
        i.remove();             // fjerner
        return true;            // vellykket fjerning
      }
    }
  }
  else                          // t er ikke lik null
  {
    while (i.hasNext())         // flere igjen
    {
      if (t.equals(i.next()))   // sammneligner
      {
        i.remove();             // fjerner
        return true;            // vellykket fjerning
      }
    }
  }
  return false;                 // mislykket fjerning
}
```

Programkode* 3.1.2 *c)

Også metoden `nullstill()` må kodes ved hjelp av metoden `remove` i iteratoren. Se *Oppgave* 2. Se *Oppgave* 3 når det gjelder metoden `toString()`.

Oppgaver til Avsnitt 3.1.2

1. Kod metoden `inneholder(T t)` i klassen AbstraktBeholder. Bruk en *forAlle*-løkke.
2. Lag kode for metoden `nullstill()` i klassen AbstraktBeholder. Bruk metoden `remove()` i iteratoren.
3. Lag kode for metoden `toString()` i klassen AbstraktBeholder. Opprett en `StringBuilder` *s* og bruk en *forAlle*-løkke til å legge verdiene fra beholderen over i *s*. Avslutt med å returnere `s.toString()`. Lag dette slik at hvis beholderen f.eks. inneholder tallene 1, 2 og 3, skal tegnstrengen bli "[1, 2, 3]". Hvis beholderen er tom skal det bli "[]".

3.1.3 En konkret beholder

I *Programkode* 3.1.2 *a)* ble det satt opp en abstrakt beholder. Den inneholder de to abstrakte metodene `leggInn(T t)` og `iterator()`. De andre metodene ble kodet ved hjelp av disse to. Det betyr at når vi skal lage en konkret beholder, er det nok å kode de to abstrakte metodene. De andre metodene arves. Dette vil imidlertid kunne føre til at noen av metodene blir ineffektive, men det vil virke.

Vår konkrete beholder lages som en subklasse til `AbstraktBeholder` og skal få navnet `TabellBeholder`. Som navnet antyder skal vi bruke en vanlig tabell som intern datastruktur. Klassen må derfor ha to variabler – en tabell og en som holder orden på antallet verdier:

```
public class TabellBeholder<T> extends AbstraktBeholder<T>
{
  private T[] a;           // en tabell
  private int antall;      // antallet verdier

  // konstruktører og metoder kommer her
}
```

Programkode* 3.1.3 *a)

Vi trenger en standardkonstruktør som oppretter en tabell med en fast startstørrelse, og en konstruktør som gir brukeren muligheten til å bestemme startstørrelsen:

```
public TabellBeholder(int størrelse)   // konstruktør
{
  a = (T[])new Object[størrelse];      // oppretter en tabell
  antall = 0;                          // foreløpig ingen verdier
}

public TabellBeholder()                // standardkonstruktør
{
  this(10);                            // 10 som startstørrelse
}
```

Programkode* 3.1.3 *b)

Det kunne også være nyttig å ha en konstruktør som lager en kopi av en annen beholder. En beholder har alltid en iterator. Det har imidlertid enhver klasse som implementerer grensesnittet *Iterable*. Det mest generelle tilfellet får vi derfor hvis vi lager en beholder som er en kopi av «itererbar» klasse:

```
public TabellBeholder(Iterable<T> itererbar)    // konstruktør
{
  for (T verdi : itererbar) leggInn(verdi);     // kopierer
}
```

Programkode* 3.1.3 *c)

Variabelen `antall` har to funksjoner. Den forteller for det første hvor mange verdier beholderen inneholder og gir for det andre posisjonen til første ledige plass i tabellen. Dermed kan metoden `leggInn(T t)` kodes slik:

```
a[antall++] = t;   // legger inn på første ledige plass og øker antall
```

Men dette vil imidlertid gå galt hvis tabellen på forhånd er full. «Trikset» er da å «utvide» tabellen. Slike «utvidelser» er ganske vanlige og det finnes derfor en ferdig metode for dette i klassen `Arrays` i `java.util`. Den heter `copyOf`.

Metoden leggInn() må først sjekke om tabellen er full og «utvide» den hvis den er det:

```
public boolean leggInn(T t)  // duplikater er tillatt
{
  // En full tabell utvides med 50%
  if (antall == a.length) a = Arrays.copyOf(a,(3*antall)/2 + 1);
  a[antall++] = t;
  return true;   // vellykket innlegging
}
```

Programkode 3.1.3 d)

Dette gir oss så langt flg. klasse:

```
public class TabellBeholder<T> extends AbstraktBeholder<T>
{
  private T[] a;           // en tabell
  private int antall;      // antallet verdier

  public TabellBeholder(int størrelse)   // konstruktør
  {
    a = (T[])new Object[størrelse];      // oppretter en tabell
    antall = 0;                          // foreløpig ingen verdier
  }

  public TabellBeholder()                // standardkonstruktør
  {
    this(10);                            // 10 som startstørrelse
  }

  public TabellBeholder(Iterable<T> itererbar)    // konstruktør
  {
    for (T verdi : itererbar) leggInn(verdi);     // kopierer
  }

  public boolean leggInn(T t)
  {
    // En full tabell utvides med 50%
    if (antall == a.length) a = Arrays.copyOf(a,(3*antall)/2 + 1);
    a[antall++] = t;
    return true;   // vellykket innlegging
  }

  public Iterator<T> iterator() { return null; }  // foreløpig kode
}
```

Programkode 3.1.3 e)

Oppgaver til Avsnitt 3.1.3

1. Sett deg inn i hvordan metoden(e) copyOf i klassen Arrays virker. Se spesielt på hva som skjer hvis oppgitt lengde er større eller mindre enn tabellens lengde.

3.1.4 Hvordan lages en iterator?

Metoden `next()` i iteratoren skal returnere en ny verdi for hvert kall. Det er ingen spesielle krav til i hvilken rekkefølge verdiene skal komme. Men siden vi her har verdiene lagret i en `tabell`, er det kanskje naturlig å hente dem fra venstre mot høyre. I figuren under har tabellen plass til 16 verdier (lengden er 16) og det er lagt inn 12 stykker. Det betyr at variabelen `antall` er lik 12.

Figur 3.1.4 a) : En tabell med 12 verdier og med plass til 16 verdier

Iterator-klassen må ha en instansvariabel denne som holder rede på hvor aktuell verdi er, dvs. den verdien som et kall på `next()` skal returnere. Vi må også passe på at denne ikke er utenfor den «lovlige» delen av tabellen, dvs. at vi må ha denne < antall. I *Figur* 3.1.4 *a)* over betyr det at vi må ha denne < 12. Det gjøres i metoden `hasNext()`. Alt dette legges i iterator-klassen *TabellBeholderIterator*:

```
// Skal være en indre klasse i class TabellBeholder
private class TabellBeholderIterator implements Iterator<T>
{
  private int denne = 0;       // instansvariabel

  public boolean hasNext()     // sjekker om det er flere igjen
  {
    return denne < antall;     // sjekker verdien til denne
  }

  public T next()              // returnerer aktuell verdi
  {
    if (!hasNext())
      throw new NoSuchElementException("Tomt eller ingen verdier igjen!");

    T temp = a[denne];         // henter aktuell verdi
    denne++;                   // flytter indeksen
    return temp;               // returnerer verdien
  }

  public void remove()
  {
    // ingen kode foreløpig
  }
}
```

Programkode 3.1.4 a)

Klassen *TabellBeholderIterator* skal ligge som en indre klasse i klassen `TabellBeholder`. Den kan f.eks. legges rett foran metoden `iterator()` og den metoden kodes slik:

```
public Iterator<T> iterator()
{
  return new TabellBeholderIterator();
}
```

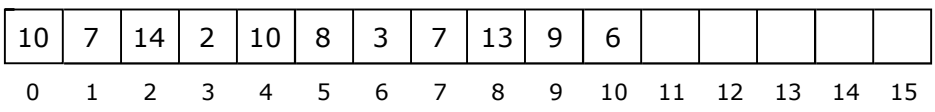
Programkode 3.1.4 b)

Nå står det igjen å kode `remove()` i klassen `TabellBeholderIterator`. Den skal fjerne verdien som ble returnert av det siste kallet på metoden `next()` og det skal ikke være mulig å kalle den to ganger på rad – se *Tabell* 3.1.1. Vi innfører derfor en ekstra instansvariabel `removeOK` i iterator-klassen. Den settes til `true` etter et kall på `next()` og til `false` etter et kall på `remove()`. Det betyr at vi må gjøre en endring i metoden `next()`:

```
private boolean removeOK = false;   // ny instansvariabel

public T next()    // ny versjon
{
  if (!hasNext())
    throw new NoSuchElementException("Tomt eller ingen verdier igjen!");

  T temp = a[denne];          // henter aktuell verdi
  denne++;                    // flytter indeksen
  removeOK = true;            // nå kan remove() kalles
  return temp;                // returnerer verdien
}

public void remove()
{
  if (!removeOK) throw
    new IllegalStateException("Ulovlig tilstand!");

  removeOK = false;           // remove() kan ikke kalles på nytt

  // verdien i posisjon denne - 1 skal fjernes siden den ble returnert
  // i det siste kallet på next(), verdiene fra og med denne flyttes
  // derfor en enhet mot venstre

  antall--;           // en verdi vil bli fjernet
  denne--;            // denne må flyttes til venstre

  for (int i = denne; i < antall; i++)
  {
    a[i] = a[i+1];    // verdiene flyttes
  }

  a[antall] = null;   // verdien som lå lengst til høyre nulles
}
```

Programkode* 3.1.4 *c)

Eksempel: Ta utgangspunkt i *Tabell* 3.1.4 *a)*. Anta at `denne = 6` (med tilhørende verdi 8). Det betyr at det er verdien i posisjon 5 (tallet 21) som skal fjernes. Det gjøres ved at alle verdiene til høyre for posisjon 5 flyttes en enhet mot venstre. Den bakerste verdien (i posisjon `antall - 1 = 11`) blir liggende igjen og må «nulles vekk» derfra. I forbindelse med flyttingen må både `denne` og `antall` oppdateres. Dermed får vi dette resultatet:



Figur 3.1.4 b) : Verdien som lå i posisjon 5 er fjernet og `denne` er nå lik 5

3.1.5 Oppsummering

Ferdige versjoner av klassene laget i *Delkapittel* 3.1 ligger på:

• Beholder.html • AbstraktBeholder.html • TabellBeholder.html

I klassen `TabellBeholder` er kun metodene `leggInn()` og `iterator()` fra grensesnittet `Beholder` kodet. De andre metodene arves fra klassen `AbstraktBeholder`. Det kan bety at en del av metodene vil være ineffektive siden de er kodet ved hjelp av iteratoren. Men de vil virke. Se flg. eksempel:

```
Beholder<Integer> beholder = new TabellBeholder<Integer>();

for (int i = 1; i <= 10; i++) beholder.leggInn(i);

System.out.println("Antall verdier: " + beholder.antall());
System.out.println(beholder);  // bruker metoden toString

System.out.println("Fjerner oddetallene: ");
for (int i = 1; i <= 10; i++)
  if (i % 2 != 0) beholder.fjern(i);  // fjerner odddetallene

System.out.println(beholder);  // bruker metoden toString

// Utskrift:
// Antall verdier: 10
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
// Fjerner oddetallene:
// [2, 4, 6, 8, 10]
```

Programkode* 3.1.5 *a)

Metoden `antall()` er kodet i `AbstraktBeholder` og finner antallet ved å telle opp ved hjelp av iteratoren. Men i `TabellBeholder` har vi en instansvariabel `antall` som hele tiden holder på antallet verdier i beholderen. Metoden `antall()` blir derfor langt mer effektiv hvis den rett og slett returnerer verdien til variabelen. Med andre ord lønner det seg å «overskrive» metoden `antall()`, dvs. lage ny kode for den i klassen `TabellBeholder`. Se *Oppgave* 1.

Vårt grensesnitt `Beholder` er en miniversjon av grensenittet `Collection` i Java. Men der er det ingen konkrete klasser som implementerer det grensesnittet. Java har også grensesnittet `Set` som tilsynelatende er helt likt `Collection`. Forskjellen er at der tillates ikke duplikater, dvs. at alle verdiene må være forskjellige.

Oppgaver til Avsnitt 3.1.5

1. Lag kode for metoden `antall()` i `TabellBeholder`. La den returnere variabelen `antall`.
2. Lag kode for metoden `nullstill()` i `TabellBeholder`. Lag den så effektiv som mulig.
3. Gjør om klassen `TabellBeholderIterator` i `TabellBeholder` slik at iteratoren starter bakerst i tabellen og går mot venstre.