

Kapittel 10: Bruk av dynamiske datastrukturer

Redigert av:

Khalid Azim Mughal (khalid@ii.uib.no)

Kilde:

Java som første programmeringsspråk (3. utgave)

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen
Cappelen Akademisk Forlag, 2006.

ISBN: 82-02-24554-0

<http://www.ii.uib.no/~khalid/jfps3u/>

(NB! Boken dekker opptil Java 6, men notatene er oppdatert til Java 7.)

Emneoversikt

Dynamiske datastrukturer

Abstrakte datatyper (ADT'er)

Dynamiske strenger: `StringBuilder`

Generiske typer

Samlinger og kontraktene `Collection`, `List` og `Set`:

- Dynamiske tabeller: `ArrayList`
- Mengder: `HashSet`

Nøkkeltabeller og kontrakten `Map`:

- Nøkkeltabeller: `HashMap`

Subtyping med jokertegn

Genereriske metoder

Dynamiske datastrukturer

- En *samling* er en datastruktur for oppbevaring av data, f.eks. tabeller definerer samlinger som har fast lengde.
- Dynamiske datastrukturer kan vokse og krympe ettersom data blir satt inn i og hentet fra strukturen.
 - *Kontrakten* angir hvilke *operasjoner* som kan utføres på strukturen.
 - Klienter trenger ikke å vite hvordan den er implementert.
- To vanligste operasjoner:
 - *Innsetting*: sette inn data i strukturen.
 - *Oppslag*: hente ut data fra strukturen.
- Valg av datastruktur er i stor grad betinget av hvor kostbart det er å utføre innsetting og oppslag.

Abstrakte datatyper: ADT = datastruktur + kontrakt

- Resultat av *data abstraksjon* er ADT'er, dvs. utforming av en *ny* type med tilhørende *data representasjon* og *operasjoner*.
- I Java er ADT'er klasser.

Dynamiske strenger: `StringBuilder`-klassen

- Innholdet av et `String`-objekt kan *ikke* endres, dvs. *tilstanden kan bare leses*.
- Java har en predefinert klasse `StringBuilder` for å håndtere *sekvenser av tegn som kan endres, og der antall tegn kan dynamisk vokse og krympe*.
- Et objekt av klassen `StringBuilder` holder rede på
 - *størrelse* (hvor mange tegn den inneholder til enhver tid), og
 - *kapasitet* (hvor mange tegn som kan innsettes i den før den blir full)
 - Dersom det ikke er plass til flere tegn, utvider den kapasiteten automatisk.
- Velg klassen `StringBuilder` fremfor klassen `String` dersom tegnsekvens endres hyppig.
- Java har støtte for *deklarasjon, opprettelse og bruk* av dynamiske strenger.

Deklarasjon:

```
StringBuilder variabelNavn;
```

- deklarasjonen oppretter en *referanse* for et `StringBuilder`-objekt.

```
StringBuilder buffer1;
```

fører til opprettelse av en referanse for et `StringBuilder`-objekt:

Navn: `buffer1`

Type: `ref(StringBuilder)`

<code>null</code>

Dynamiske strenger: Opprettelse

- `StringBuilder`-objektet kan opprettes ved å anvende `new`-operatoren på en `StringBuilder`-konstruktør.
- Vi kan kombinere deklarasjon med opprettelse:

```
StringBuilder strengbygger = new StringBuilder(argument-liste);
```

```
// Oppretter et StringBuilder-objekt som har ingen tegn og lengde 0,  
// men kapasitet på 16:
```

```
StringBuilder navnBuffer = new StringBuilder();
```

```
// Oppretter et StringBuilder-objekt som har ingen tegn og lengde 0,  
// men kapasitet på 10:
```

```
StringBuilder adrBuffer = new StringBuilder(10);
```

```
// Oppretter et StringBuilder-objekt fra en strenglitteral,  
// med kapasitet på strenglengde + 16, dvs 20 tegn i dette tilfellet:
```

```
StringBuilder fargeBuffer = new StringBuilder("rødt");
```

```
// Oppretter et StringBuilder-objekt fra et String-objekt.
```

```
StringBuilder strBuffer = new StringBuilder(str);
```

Operasjoner på strengbufferne

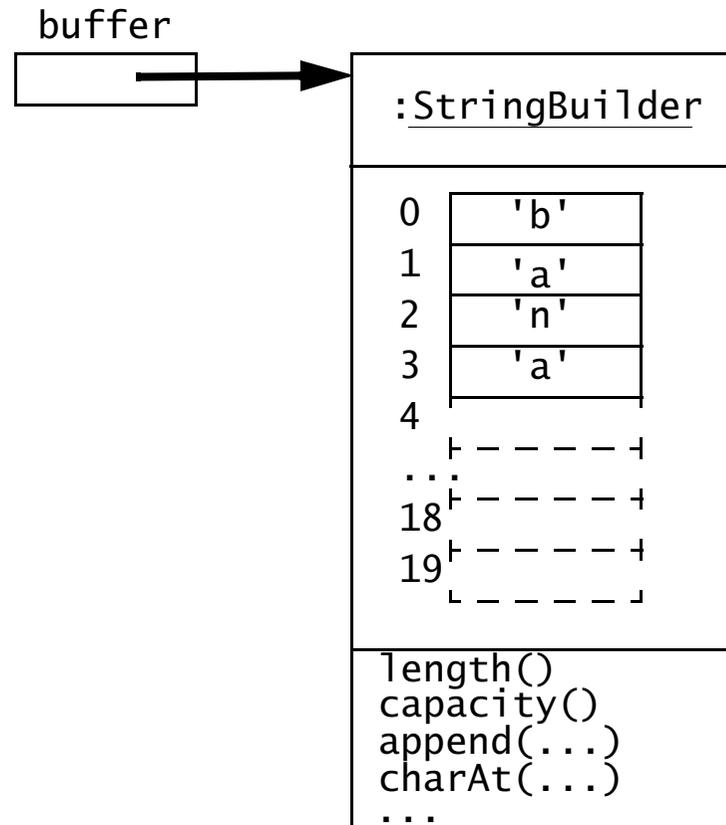
```
StringBuilder kursBuffer = new StringBuilder("databelhandling");
```

Selektorer for StringBuilder-klassen:

- Hvert StringBuilder-objekt har en instansmetode, `length()`, som returnerer antall tegn i strengbufferen (*størrelse*).
 - Metodekall `kursBuffer.length()` returnerer antall tegn i StringBuilder-objektet `kursBuffer`, dvs. 14.
- Hvert StringBuilder-objekt har en instansmetode, `capacity()`, som returnerer antall tegn som kan settes inn i strengbufferen før den utvides for å lagre flere tegn (*kapasitet*).
 - Metodekall `kursBuffer.capacity()` returnerer antall tegn som kan lagres totalt i StringBuilder-objekt `kursBuffer`, dvs. 30.
- Metoden `charAt(int i)` returnerer tegn gitt ved indeks `i` i strengen.
 - Metodekall `kursBuffer.charAt(4)` returnerer tegn 'b' i StringBuilder-objekt `kursBuffer`.
 - Startindeks er 0.
 - Ulovlig indeksverdi resulterer i `StringIndexOutOfBoundsException`.

StringBuilder-objekt

```
StringBuilder buffer = new StringBuilder("bana");
```



StringBuilder-objektet har 4 tegn innsatt, og plass til 16 til - og kan vokse dersom det er nødvendig.

Operasjoner på strengbyggere (forts.)

Modifikatorer for StringBuilder-klassen:

- Tegn kan innsettes *hvor som helst* i strengbyggeren.
 - Innsetting kan føre til at andre tegn flyttes for å gjøre plass til det nye tegnet.
 - Størrelsen justeres automatisk ved innsetting.
- Den overlastede metoden `append()` kan brukes til å tilføye primitive verdier, String-objekter, tabeller av tegn og tekstrepresentasjon av andre objekter på *slutten* av strengbyggeren.

```
StringBuilder buffer = new StringBuilder("bana");  
buffer.append("na"); // tilføyer en streng på slutten av strengbyggeren: "banana"  
buffer.append(42); // tilføyer et tall på slutten av strengbyggeren: "banana42"
```

```
StringBuilder strBuffer = new StringBuilder().append(4).append("U").append("Only");  
String str = 4 + "U" + "Only"; // bruker en StringBuilder implisitt
```

- Den overlastede metoden `insert()` kan brukes til å innsette primitive verdier, String-objekter, tabeller av tegn og andre objekter ved en gitt *indeks* i strengbyggeren.

```
buffer.insert(6, "Rama"); // "bananaRama42"  
buffer.insert(11, 'U'); // "bananaRama4U2"  
buffer.setCharAt(6, 'm'); // "bananamama4U2"
```
- `StringBuilderKlient.java` viser flere operasjoner på strengbygge re.

Operasjoner på strengbyggere (forts.)

- Klassen `StringBuilder` overkjører *ikke* `equals()`-metoden fra `Object`-klassen.
- Strengbyggere må oversettes til strenger for å sammenligne dem:

```
boolean status = buffer1.toString().equals(buffer2.toString());
```

Generiske typer

- ADT'er der vi kan bytte ut *referansetyper*, kalles for *generiske typer*.
- Første utkast av par med verdier:

```
class ParObj {
    private Object første;
    private Object andre;
    ParObj () { }
    ParObj (Object første, Object andre) {
        this.første = første;
        this.andre = andre;
    }
    public Object hentFørste() { return første; }
    public Object hentAndre() { return andre; }
    public void settFørste(Object nyFørste) { første = nyFørste; }
    public void settAndre(Object nyAndre) { andre = nyAndre; }
}
}
```

- Klient av klassen ParObj:

```
class ParObjKlient {
    public static void main(String[] args) {
        ParObj førstePar = new ParObj("Adam", "Eva"); // Kan innsette hva vi vil.
        ParObj etPar = new ParObj("17.mai", 1905);
        Object obj = førstePar.hentFørste();
        if (obj instanceof String) {                // Er objektet av riktig type?
            String str = (String) obj;              // Typekonvertering til subklasse.
            System.out.println(str.toLowerCase()); // Spesifikk metode i subklassen.
        }
    }
}
```

- Klient må holde rede hva som settes inn i et ParObj.
- Krever sjekking og typekonvertering ved oppslag.

Generiske klasser

- En generisk klasse som kan brukes til å lage par av objekter der begge objektene har samme type:

```
class Par<T> { // (1)
    private T første;
    private T andre;
    Par () { }
    Par (T første, T andre) {
        this.første = første;
        this.andre = andre;
    }
    public T hentFørste() { return første; }
    public T hentAndre() { return andre; }
    public void settFørste(T nyFørste) { første = nyFørste; }
    public void settAndre(T nyAndre) { andre = nyAndre; }

    @Override
    public String toString() {
        return "(" + første.toString() + "," + andre.toString() + ")"; // (2)
    }
}
```

- En generisk klasse spesifiser en eller flere *formelle typeparametere*, f.eks. $\langle T \rangle$.
 - I den generiske klassen $\text{Par}\langle T \rangle$ har vi brukt T alle steder hvor vi brukte typen `Object` i definisjon av klassen `ParObj`.
 - Typeparameteren er brukt som en vanlig referansetype i klassekroppen: som feltype, som returtype og som parametertype i metodene.
 - Hvilken referansetype typeparameteren T egentlig står for er ikke kjent i den generiske klassen $\text{Par}\langle T \rangle$.
- Konstruktører tar ikke formelle typeparametere.

Parametriserte typer

- En generisk klasse brukes ved å angi *aktuelle typeparametere* som erstatter de *formelle typeparametere* i klassedefinisjonen under kompilering.
- F.eks. vil `Par<String>` introdusere en *ny referansetype under kompilering*, dvs. `par` som bare tillater `String`-objekter, der den formelle typeparameteren `T` er erstattet av den aktuelle typeparameteren `String`.
- Kompilatoren kontrollerer at parametriserte typer blir korrekt brukt i kildekoden, slik at det ikke oppstår kjørefeil.
- Aktuelle typeparametere angis etter klassenavnet, på samme måte som formelle typeparametere i en generisk klassedefinisjon.
- Primitive datatyper kan ikke angis som aktuelle typeparametere.
- Forholdet mellom generiske typer (`Par<T>`) og parametriserte typer (`Par<String>`) kan sammenlignes med forholdet mellom deklarasjon og kall av en metode.

```

public class ParametriserteTyper {

    public static void main(String[] args) {
        Par<String> strPar = new Par<String>("Adam", "Eva"); // (1)
        // Par<String> feilPar = new Par<String>("17.mai", 1905); // (2) Kompilerings-
                                                // feil!
        Par<Integer> intPar = new Par<Integer>(2005, 2010); // (3)
        // strPar = intPar; // (4) Kompileringsfeil!
        Par<String> tempPar = strPar; // (5) OK

        strPar.settFørste("Ole"); // (6) OK. Kun String godtatt.
        // intPar.settAndre("Maria"); // (7) Kompileringsfeil! Kun Integer godtatt.
        String navn = strPar.hentAndre().toLowerCase(); // (8) "eva"
        System.out.println(navn);
    }
}

```

- Klient trenger ikke holde rede hva som settes inn i et Par.
- Ikke nødvendig med sjekking og typekonvertering ved oppslag.

Generiske kontrakter

- Eksempel:

```
interface ParForhold<T> {  
    T hentFørste();  
    T hentAndre();  
    void settFørste(T nyFørste);  
    void settAndre(T nyAndre);  
}
```

- En generisk kontrakt kan implementeres av en generisk (eller en ikke-generisk) klasse:

```
class Par<T> implements ParForhold<T> {  
    // samme som før  
}
```

- Vi kan deklarere referanser av parameteriserte kontrakter.

```
ParForhold<String> etStrPar = new Par<String>("Eva", "Adam"); // (9)
```

– Par<String> er en subtype av ParForhold<String>.

- Fra Java-standardbiblioteket:

```
public interface Comparable<T> {  
    int compareTo(T obj);  
}
```

- En klasse som vil angi en naturlig ordning for sine objekter, kan implementere Comparable<T>- kontrakten:

```
class Dings implements Comparable<Dings> {  
    public int compareTo(Dings dings2) { /* Implementering */ }  
    // ...  
}
```

- Merk at vi har parametrisert Comparable<T> med Dings, siden det er dingser metoden compareTo() skal sammenligne i klassen Dings.

Generiske typer under kompilering

- Den generiske klassen `Par<T>` blir oversatt og blir representert ved klassen `Par`, dvs én class-fil (`Par.class`) med Java-bytekode.
- Parametriserte typer brukes av kompilatoren for å kontrollere at objekter som opprettes blir brukt riktig i programmet.
- Kjøremiljøet er derimot uvitende om bruk av generiske typer, dvs. den forholder seg kun til klassen `Par`.
- Siden kun én klasse representerer alle parametriseringer av en generisk klasse, og bare én forekomst av et statisk medlem kan eksistere i en klasse, kan ikke statiske metoder referere til formelle typeparametere.
- Kompilatoren gir advarsel (eng. *unchecked warning*) i tilfeller der bruk av en generisk type uten typeparametere kan skape problemer under utføring.

Diamant-notasjon (<>) i deklarasjonsetning

- Tidligere har vi deklarerert en variabel og opprettet et objekt av en parametrisert type slik:

```
Par<String> strPar = new Par<String>("Adam", "Eva"); // (1)
```

- I denne konteksten, kan vi bruke diamant-notasjon:

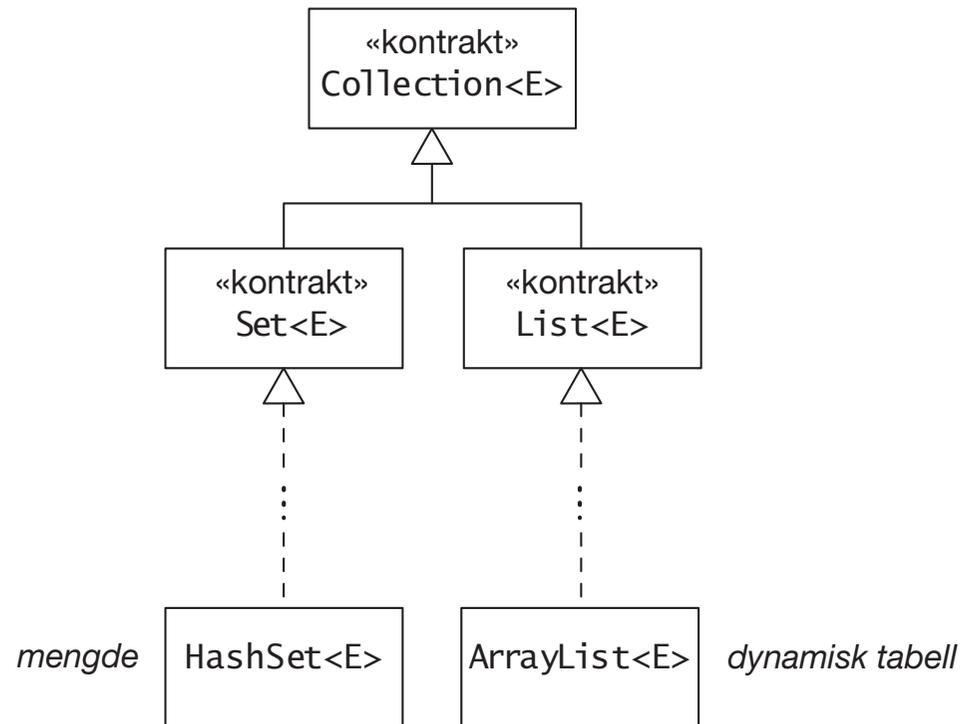
```
Par<String> strPar = new Par<>("Adam", "Eva"); // Bruk av <>.
```

– Vi kan utelate parameter typene i konstruktørkallet i en deklarasjonsetning.

- Kompilatoren klarer å tolke at objektet som opprettes skal ha type `Par<String>`, og det er kompatibel med typen til variabelen på den venstre siden.

Samlinger

- En *samling* er en datastruktur som kan holde rede på referanser til andre objekter.
 - For eksempel, en tabell med referanser til objekter er en samling.
- Java API definerer flere andre typer samlinger i `java.util`-pakken.
- Sentralt i `java.util`-pakken er noen få viktige *generiske kontrakter* som samlingene implementerer.



Kontrakten Collection<E>

- *Basisoperasjoner* er de vanligste som utføres på samlinger: innsetting, sletting og kontroll av medlemskap.

Utvalgte basisoperasjoner fra kontrakten Collection<E>

<code>int size()</code>	Returnerer antall elementer i samlingen.
<code>boolean isEmpty()</code>	Finner ut om samlingen er tom.
<code>boolean contains(Object element)</code>	Finner ut om element er med i samlingen.
<code>boolean add(E element)</code>	<i>Innsetting</i> : prøver å tilføye element til samlingen og returnerer sann dersom dette lykkes.
<code>boolean remove(Object element)</code>	<i>Sletting</i> : prøver å slette element fra samlingen og returnerer sann dersom dette lykkes.
<code>Iterator<E> iterator()</code>	Returnerer en iterator som kan brukes til å gjennomløpe samlingen.

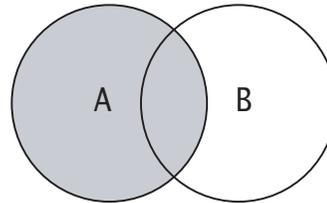
- *Bulkoperasjoner* utføres på hele samlinger.

Utvalgte bulkoperasjoner fra kontrakten `Collection<E>`

<code>boolean containsAll(Collection<?> s)</code>	<i>Delmengde</i> : returnerer sann dersom alle elementer i samling <code>s</code> er i denne samlingen.
<code>boolean addAll(Collection<? extends E> s)</code>	<i>Union</i> : tilføyer alle elementer fra samlingen <code>s</code> til denne samlingen og returnerer sann dersom denne samlingen ble endret.
<code>boolean retainAll(Collection<?> s)</code>	<i>Snitt</i> : beholder i denne samlingen kun de elementene som også er i samlingen <code>s</code> , og returnerer sann dersom denne samlingen ble endret.
<code>boolean removeAll(Collection<?> s)</code>	<i>Differanse</i> : sletter alle elementer i denne samlingen som også er i samlingen <code>s</code> , og returnerer sann dersom denne samlingen ble endret.
<code>void clear()</code>	Sletter alle elementer fra denne samlingen.

- Merk at metodene `addAll()`, `retainAll()` og `removeAll()` er destruktive, dvs at de kan endre den aktuelle samlingen.

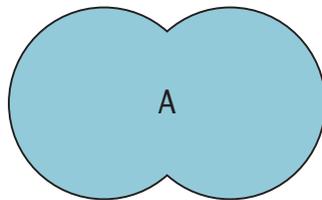
Mengdelæren



A = [kiss, Sivle, madonna, aha, abba]

B = [TLC, wham, madonna, abba]

(a)



A.addAll(B)

Etter utføring:

A = [kiss, TLC, Sivle, wham, aha, madonna, abba]

(b) Union

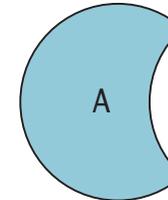


A.retainAll(B)

Etter utføring:

A = [madonna, abba]

(c) Snitt



A.removeAll(B)

Etter utføring:

A = [kiss, Sivle, aha]

(d) Differanse

Gjennomløpe en samling

- Alle samlinger implementer:

```
interface Iterable<E> {  
    Iterator<E> iterator();  
}
```

- En for(:)-løkke kan brukes til å traversere en samling:

```
// Opprett en dynamisk tabell.
```

```
Collection<String> samling = new ArrayList<>();
```

```
    // Legg til elementer.
```

```
samling.add("9");
```

```
samling.add("1");
```

```
samling.add("1");
```

```
Iterator<String> iter = samling.iterator(); // Hent iterator.
```

```
while (iter.hasNext()) { // Flere elementer i samlingen?
```

```
    System.out.print(iter.next()); // Hent inneværende element.
```

```
}
```

```
// Traverser samlingen med for(:)-løkke.
```

```
for (String element: samling) {
```

```
    System.out.print(element); // Håndter inneværende element.
```

```
}
```

- En *iterator* er et objekt som gjør det mulig å behandle alle objekter i samlingen i tur og orden på en systematisk måte (Tabell 9.5).

Operasjoner i kontrakten Iterator	
<code>boolean hasNext()</code>	Returnerer sann dersom den underliggende samlingen har elementer igjen å behandle.
<code>E next()</code>	Returnerer neste element i den underliggende samlingen og inkrementerer iteratoren. Dersom det ikke er flere elementer å returnere, vil kallet til denne metoden resultere i et unntak av typen <code>NoSuchElementException</code> .
<code>void remove()</code>	Fjerner det inneværende elementet fra den underliggende samlingen.

- Legg merke til at disse metodene kalles på *iteratoren*, ikke på samlingen:

```
// Opprett en dynamisk tabell.
```

```
Collection<String> samling = new ArrayList<>();
```

```
samling.add("9"); // Legg til elementer.
```

```
samling.add("1");
```

```
samling.add("1");
```

```
Iterator<String> iter = samling.iterator(); // Hent iterator.
```

```
while (iter.hasNext()) { // Flere elementer i samlingen?
```

```
    System.out.print(iter.next()); // Hent inneværende element.
```

```
}
```

- Metoden `hasNext()` og metoden `next()` brukes *i takt*, ellers kan det oppstå feil under kjøring dersom metoden `next()` kalles og det er ikke flere elementer igjen.
- Standard strengrepresentasjon for en samling er:
 $[element_0, element_1, \dots, element_{n-1}]$
 - Metoden `toString()` vil returnere denne strengrepresentasjon.

Dynamiske tabeller: ArrayList<E>

Subkontrakten List

- En samling som tillater duplikater, og hvor elementene er *ordnet*, kalles en *liste*.
- Kontrakten List utvider kontrakten Collection til å gjelde for lister.
- Elementene har en *posisjon* (angitt ved en *indeks*) i listen.

Utvalgte listeoperasjoner i kontrakten List

E get(int indeks)	Returnerer element gitt ved indeks.
E set(int indeks, E element)	Erstatter elementet gitt ved indeks med element. Returnerer elementet som ble erstattet.
void add(int indeks, E element)	Innsetter element i angitt indeks. Elementene forskyves dersom det er nødvendig.
E remove(int indeks)	Sletter og returnerer element i angitt indeks. Elementene forskyves dersom det er nødvendig.
int indexOf(Object obj)	Returner indeks til første forekomst av obj dersom objektet finnes, ellers -1.

Dynamiske tabeller (DynamiskTabKlient.java)

- Programmet leser argumenter fra kommandolinjen inn i en dynamisk tabell (ordListe), og *sladder ut* spesifikke ord fra denne listen.
- Vi bruker en import-setning i begynnelsen av kildekodefilen, slik at vi slipper å bruke hele pakkestien for å referere til klasser fra java.util-pakken.
- Ord som skal sladdes, er gitt i en egen dynamisk tabell (sladdeteOrd).
- Ved (1) opprettes en tom dynamisk tabell for strenger:
`List<String> ordListe = new ArrayList<>();`
- Ved (2) konstrueres det en dynamisk tabell (sladdeteOrd) med ord som skal sladdes.
`List<String> sladdeteOrd = new ArrayList<>();`
- Ved (3) bruker vi en for(:)-løkke på ordlisten for å kontrollere hvert ord i den.

```
for (String ord : ordListe) {  
    if (sladdeteOrd.indexOf(ord) != -1) {  
        int indeksIOrdlisten = ordListe.indexOf(ord);  
        ordListe.set(indeksIOrdlisten, SENSUR);  
    }  
}
```

Mengder: HashSet<E>

Subkontrakten Set<E>

- Kontrakten Set modellerer det matematiske mengdebegrepet som tillater operasjoner som *union*, *snitt* og *differanse* fra mengdelæren.
- En mengde tillater *ikke* duplikate verdier.
- Kontrakten Set introduserer ikke nye metoder utover dem som allerede finnes i kontrakten Collection, men spesialisierer disse for mengder.

Eksempel: Mengder (MengdeKlient.java)

- Eksempel: To mengder med artistnavn.
- En tom mengde for konsert A oppretter vi ved å instansiere klassen HashSet:

```
Set<String> konsertA = new HashSet<>();
```
- Vi tilføyer de forskjellige artistene ved å bruke add()-metoden:

```
konsertA.add("aha");
```

 - Dersom add()-metoden førte til at et element ble tilføyet mengden, returneres sannhetsverdien true.
 - Sannhetsverdien false vil indikere at elementet allerede finnes i mengden.
- Koden nedenfor viser hvordan vi kan lage en kopi av en mengde ved å kalle riktig konstruktør i klassen HashSet<E>:

```
Set<String> alleArtister = new HashSet<>(konsertA);
```

Eksempel: Lage en dynamisk tabell uten duplikater (Duplikater.java)

- Koden nedenfor vil lage en mengde (ordMengde) av String-objekter fra en dynamisk tabell (ordListe) av String-objekter og dermed fjerne duplikater:

```
import java.util.*;

public class Duplikater {
    public static void main(String args[]) {
        ArrayList<String> ordListe = new ArrayList<>(); // Opprinnelig liste
        ordListe.add("to"); ordListe.add("null");
        ordListe.add("null"); ordListe.add("fem");
        System.out.println("Opprinnelig ordliste: " + ordListe);

        Set<String> ordMengde = new HashSet<>(ordListe); // Ny mengde
        ordListe = new ArrayList<>(ordMengde); // Listen uten duplikater.
        System.out.println("Opprinnelig ordliste, ingen duplikater: " + ordListe);
    }
}
```

Utdata fra programmet:

Opprinnelig ordliste: [to, null, null, fem]

Opprinnelig ordliste, ingen duplikater: [to, null, fem]

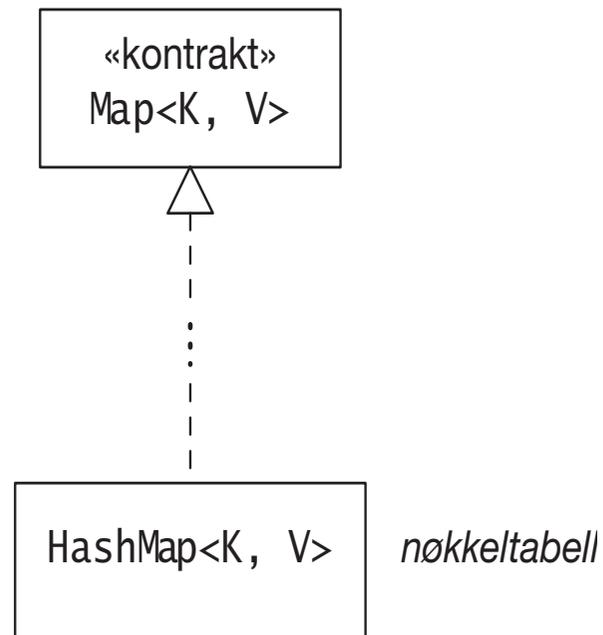
Nøkkeltabeller: Map

- En *nøkkeltabell* brukes til å lagre *bindinger*.
- En binding er et par av objekter, der det ene objektet (kalt *nøkkel*) er assosiert med det andre objektet (kalt *verdi*).
- Eksempel: En telefonliste er en nøkkeltabell, der hver binding assosierer et telefonnummer (nøkkel) med et navn (verdi).
- Det er en *mange-til-en*-relasjon mellom nøkler og verdier i en nøkkeltabell:
 - Forskjellige nøkler kan ha samme verdi, men forskjellige verdier kan ikke ha samme nøkkel.
 - dvs. nøkler er entydige i en nøkkeltabell.

Kontrakten Map<K, V>

- Funksjonaliteten til nøkkeltabeller er spesifisert i kontrakten Map i `java.util`-pakken.
- Klassen `HashMap` er en konkret implementering av kontrakten Map.
- Vi kan opprette en tom nøkkeltabell for bindinger `<String, Integer>` slik:

```
Map<String, Integer> ordTabell = new HashMap<>();
```



Basisoperasjoner i kontrakten Map<K, V>

Utvalgte basisoperasjoner fra kontrakten Map<K, V>

<code>int size()</code>	Returnerer antall bindinger i nøkkeltabellen.
<code>boolean isEmpty()</code>	Returnerer sann dersom nøkkeltabellen ikke har noen bindinger.
<code>V put(K nøkkel, V verdi)</code>	Binder <code>nøkkel</code> til verdi og lagrer bindingen. Dersom <code>nøkkel</code> hadde en tidligere binding, returneres verdien fra den tidligere bindingen.
<code>V get(Object nøkkel)</code>	Returnerer verdien fra bindingen som <code>nøkkel</code> har, dersom <code>nøkkel</code> var registrert fra før. Ellers returneres referanseverdien <code>null</code> .
<code>V remove(Object nøkkel)</code>	Prøver å slette bindingen til <code>nøkkel</code> og returnerer verdien i denne bindingen dersom <code>nøkkel</code> var registrert fra før.
<code>boolean containsKey(Object nøkkel)</code>	Returnerer sann dersom <code>nøkkel</code> har en binding.
<code>boolean containsValue(Object verdi)</code>	Returnerer sann dersom verdi er med i minst én binding.

Hashing

- Lagring og oppslag på bindinger i en nøkkeltabell krever at det er mulig å identifisere en nøkkel i en binding ved hjelp av et heltall, kalt *hashkode*.
- Java bruker metoden `hashCode()` til å beregne en hashkode for et objekt, som er definert i klassen `Object`.
- Hashkoden må oppfylle følgende betingelser:
 - Hashkoden skal alltid være den samme for et objekt, så lenge tilstanden til objektet ikke er endret.
 - To objekter som er identiske ifølge `equals()`-metoden må også ha samme hashkode.
- Klassen `String` og wrapper-klasser overkjører derfor både `hashCode()`- og `equals()`-metodene fra `Object`-klassen.

Hovedregel: Dersom en klasse overkjører `equals()`, bør den også overkjører `hashCode()`.

Overkjøring av hashCode()-metoden

- Klassen Punkt3D_V1 overkjører hverken equals()-metoden eller hashCode()-metoden:

```
class Punkt3D_V1 { // (1)
    int x;
    int y;
    int z;

    Punkt3D_V1(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    @Override
    public String toString() { return "["+x+", "+y+", "+z+"]"; }
}
```

- Klassen Punkt3D overkjører både equals()- og hashCode()-metoden:

```
class Punkt3D { // (2)
    int x;
    int y;
    int z;
    // ...
    @Override
    public boolean equals(Object obj) { // (3)
        if (this == obj) return true;
        if (!(obj instanceof Punkt3D)) return false;
        Punkt3D p2 = (Punkt3D) obj;
        return this.x == p2.x && this.y == p2.y && this.z == p2.z;
    }
    @Override
    public int hashCode() { // (4)
        int hashkode = 11;
        hashkode = 31 * hashkode + x;
        hashkode = 31 * hashkode + y;
        hashkode = 31 * hashkode + z;
        return hashkode;
    }
}
```

- Klassen Hashing2 viser hva som skjer når vi bruker punkt-objekter:

Når equals()- og hashCode()-metodene ikke er overkjørt:

```
Punkt3D_V1 p1[1,2,3]: 11394033
```

```
Punkt3D_V1 p2[1,2,3]: 4384790
```

```
p1.hashCode() == p2.hashCode(): false
```

```
p1.equals(p2): false
```

```
Nøkkeltabell med Punkt3D_V1: {[1,2,3]=2, [1,2,3]=5} <=== Nøkler er ikke entydige.
```

```
Verdi for [1,2,3]: null
```

```
<=== Kan ikke finne nøkkelen.
```

Når equals()- og hashCode()-metodene er overkjørt:

```
Punkt3D pp1[1,2,3]: 328727
```

```
Punkt3D pp2[1,2,3]: 328727
```

```
pp1.hashCode() == pp2.hashCode(): true
```

```
pp1.equals(pp2): true
```

```
Nøkkeltabell med Punkt3D: {[1,2,3]=5}
```

```
Verdi for [1,2,3]: 5
```

```
<=== Nøkler er entydige.
```

```
<=== Kan finne nøkkelen.
```

Eksempel: Basisoperasjoner i kontrakten Map<K, V>

Ord (nøkkel)	Hyppighet (verdi)
da	3
var	2
det	2
og	1

- Metoden put() lager en binding og setter den inn i nøkkeltabellen:
`ordTabell.put("da", 3); // <"da", new Integer(3)>`
 - Metoden vil *overskrive* en tidligere binding til nøkkelen og returnere verdien fra den tidligere bindingen.
- Hente ut verdien til en nøkkel:
`int hyppighet = ordTabell.get("da"); // 3`
- Binding til en nøkkel kan slettes med remove()-metoden som returnerer verdien i bindingen.
- Om en verdi forekommer i én eller flere bindinger:
`boolean nøkkelFinnes = ordTabell.containsKey("da"); // sann`
`boolean verdiFinnes = ordTabell.containsValue(2001); // usann`

- Standard strengrepresentasjon av en nøkkeltabell:
`{da=3, var=2, det=2, og=1}`

Nøkkeltabellutsnitt

- Et *nøkkeltabellutsnitt* er en samling som er assosiert med en underliggende nøkkeltabell.
- Ved hjelp av et slikt utsnitt kan vi for eksempel iterere over den underliggende nøkkeltabellen.
- Endringene som gjøres via et utsnitt, blir reflektert i den underliggende nøkkeltabellen.

Utvalgte utsnittsoperasjoner fra kontrakten Map

<code>public Set<K> keySet()</code>	<i>(Nøkkelutsnitt)</i> Returnerer Set-utsnitt av alle <i>nøkler</i> i nøkkeltabellen.
<code>public Collection<V> values()</code>	<i>(Verdiutsnitt)</i> Returnerer Collection-utsnitt av alle <i>verdier</i> i alle bindinger i nøkkeltabellen.

- *Nøkkelutsnitt:*

```
Set<String> ordMengde = ordTabell.keySet(); // [da, var, det, og]
```

- Siden nøkler er entydige, er det hensiktsmessig å lage en mengde som ikke tillater duplikater.
- En `for(:)`-løkke kan brukes til å iterere over nøkler i denne mengden på vanlig måte.
- Metoden `get()` kan brukes på nøkkeltabellen for å hente ut den tilsvarende verdien til nøkkelen.

- *Verdiutsnitt:*

```
Collection<Integer> antallSamling = ordTabell.values(); // [3, 2, 2, 1]
```

- Siden verdier ikke er entydige, er det hensiktsmessig å lage en samling som tillater duplikater.
- En `for(:)`-løkke kan brukes til å iterere over verdier i denne samlingen på vanlig måte.

Bruk av nøkkeltabeller (NoekkelTabKlient.java)

- Lesing av ord og innsetting i nøkkeltabellen:

Gjenta mens flere argumenter i kommandolinjen:

Gjør oppslag i nøkkeltabellen med inneværende argument.

Hvis inneværende argument er ikke registrert:

La hyppighet til inneværende argument være 1, dvs. 1. gang;

ellers:

Øk hyppighet til inneværende argument med 1.

Sett inn inneværende argument med riktig hyppighet.

```
// (2) Les ord fra kommandolinjen.  
for (String arg : args) {  
    // Oppslag for å finne om et ord er registrert.  
    Integer antallGanger = ordTabell.get(arg);  
    if (antallGanger == null)  
        antallGanger = 1;        // Ikke registrert før.Første gang.  
    else  
        // Registrert. Teller økes med 1.  
        antallGanger++;  
    ordTabell.put(arg, antallGanger);  
}
```

- Beregn antall ord som ble lest (4).
 - Totalt antall ord innlest er lik summen av alle hyppigheter.
 - Vi lager et *verdiutsnitt*, som brukes til å iterere over denne samlingen for å summere hyppigheter.

```
Collection<Integer> antallSamling = ordTabell.values();
int totalAntallOrd = 0;
for (Integer antall : antallSamling) {
    totalAntallOrd += antall;
}
```

- Finn alle distinkte ord i nøkkeltabellen (5):
 - Alle distinkte ord utgjør alle nøkler som er registrert.
 - Lag et *nøkkelutsnitt* som er en mengde av alle ord (det vil si nøkler) fra nøkkeltabellen, og skriver den ut.

```
Set<String> ordMengde = ordTabell.keySet();
```

- Finn alle ord som er dupliserte (6):

Lag en tom mengde for dupliserte ord.

Lag et nøkkelutsnitt som er en mengde over alle nøkler.

Gjenta mens flere elementer i mengden med alle nøkler:

 Gjør oppslag i nøkkeltabellen med inneværende nøkkel.

 Hvis hyppighet ikke er lik 1, dvs. duplisert ord:

 Innsett ord i mengde over dupliserte ord.

```
Collection<String> duplisertMengde = new HashSet<>();
for (String nøkkel : ordMengde) { // ordMengde er et nøkkelutsnitt.
    int antallGanger = ordTabell.get(nøkkel);
    if (antallGanger != 1)
        duplisertMengde.add(nøkkel);
}
```

Subtyping med jokertegn

```
static double leggSammen(Par<Number> par) {  
    return par.hentFørste().doubleValue() + par.hentAndre().doubleValue();  
}
```

...

```
double sum = leggSammen(new Par<Integer>(100, 200)); // (1) Kompileringsfeil!
```

- Par<Integer> ikke er en *subtype* til Par<Number>, jfr. tabelltypen Number[] er supertypen til tabelltypen Integer[].
- Den parametriserte typen Par<? extends T>:

```
static double leggSammen(Par<? extends Number> par) {  
    return par.hentDings1().doubleValue() + par.hentDings2().doubleValue();  
}
```

- Den parametriserte typen Par<? extends Number> står for alle par med elementtype som er enten Number eller *en subtype av* Number, mao. er Par<? extends Number> *supertypen til* Par<Number>, Par<Double> og Par<Integer>.

```
Par<Double> doublePar = new Par<>(100.50, 200.50);
```

```
double nySum = leggSammen(doublePar); // (2) Ok
```

```
Par<Number> numPar = doublePar; // (3) Kompileringsfeil!
```

```
Par<? extends Number> nyttPar = doublePar; // (4) Ok
```

- Den parametriserte typen `Par<? super Integer>`:
 - Står for alle par med elementtype som er enten `Integer` eller *en supertype av* `Integer`, mao. er `Par<? super Number>` *supertypen til* `Par<Number>`, `Par<Comparable>` og `Par<Integer>`.

```

Par<Number> tallPar = new Par<>(100.0, 200);
Par<Integer> iPar = new Par<>(100, 200);
Par<? super Integer> supPar = tallPar;           // (5) Ok
supPar = iPar;                                  // (6) Ok
supPar = doublePar;                             // (7) Kompileringsfeil!

```

- Den parametriserte typen `Par<?>`:
 - `Par<?>` representerer et hvilket som helst par.
 - `Par<?>` er supertypen for alle parametriserte typer til `Par<T>`, mao. er `Par<?>` *supertypen til* `Par<? extends Number>`, `Par<? super Integer>`, `Par<Number>` og `Par<Integer>`.

```

Par<?> parB = tallPar;                           // (8) Ok
parB = nyttPar;                                  // (9) Ok
parB = supPar;                                   // (10) Ok
parB = new Par<?>(100.0, 200);                   // (11) Kompileringsfeil!
parB = new Par<? extends Number>(100.0, 200);   // (12) Kompileringsfeil!

```

Generiske metoder

- En metode kan deklarerere sine egne formelle typeparametere og benytte dem i metoden.

```
public static <T> List<T> tabellTilListe(T[] tabell) { // (1)
    List<T> liste = new ArrayList<>(); // Opprett en tom liste.
    for (T data : tabell)             // Gjennomløp tabellen.
        liste.add(data);              // Sett et element fra tabellen inn i listen.
    return liste;                     // Returner listen.
}
```

- I en generisk metode er en formell typeparameter T bare tilgjengelig i selve metoden.
- Det er ikke et krav at en generisk metode må deklarereres i en generisk klasse.
- Vi kan kalle en generisk metode på vanlig måte.
 - Kompilatoren avgjør den aktuelle typeparameteren fra metodekallet:

```
String[] strTab = {":-(", ";-)", ":-)"};
List<String> strListe = tabellTilListe(strTab); // (2) T er String.
Integer[] intTab = {2005, 9, 26};
List<Integer> intListe = tabellTilListe(intTab); // (3) T er Integer.
```

Flere eksempel på generiske metoder

- Metoden `sorterVedInnsetting()` sorterer en liste ifølge det angitte `Comparator`-objektet (se filen `SamlingUtil.java`):

```
static <E> void sorterVedInnsetting(List<E> liste, Comparator<E> comp) { ... }
```

- Metoden `binsøk()` finner indeksen til en nøkkel i en tabell, dersom nøkkelen finnes i tabellen (se filen `TabellUtil.java`):

```
static <T extends Comparable<T>> int binsøk(T[] tab, T nøkkel) { ... }
```

- Filene `TabellUtil.java` og `SamlingUtil.java` inneholder andre nyttige metoder som er implementert som generiske metoder.

Utvalgte generiske sortering- og søkemetoder i Java API

java.util.Collections

```
static <T> int binarySearch(List<? extends Comparable<? super T>> liste, T nøkkel)
```

```
static <T> int binarySearch(List<? extends T> liste, T nøkkel,  
                           Comparator<? super T> sammenligner)
```

```
static <T> void sort(List<T> liste)
```

```
static <T> void sort(List<T> liste, Comparator<? super T> sammenligner)
```

java.util.Arrays

```
static <T> int binarySearch(T[] tabell, T nøkkel, Comparator<? super T> sammenligner)
```

```
static <T> void sort(T[] tabell, Comparator<? super T> sammenligner)
```

```
static <T> List<T> asList(T... argumenter)
```