1.4 Generiske algoritmer

1.4.1 Maks-metoder for *double* og *String*

En metode som finner posisjonen til det største tallet i en desimaltallstabell, kan lages på samme måte som vi tidligere har gjort for heltall (se *Programkode* 1.1.4):

```
public static int maks(double[] a)     // legges i class Tabell
{
  int m = 0;                           // indeks til største verdi
  double maksverdi = a[0];             // største verdi

  for (int i = 1; i < a.length; i++) if (a[i] > maksverdi)
  {
    maksverdi = a[i];     // største verdi oppdateres
    m = i;                // indeks til største verdi oppdaters
  }
  return m;     // returnerer posisjonen til største verdi
}
```

Programkode* 1.4.1 *a)

Koden er så og si identisk med den i *Programkode* 1.1.4. Det er kun gjort endringer (skrevet med rødt) i forbindelse med datatypen. Der det refereres til tabellindekser endrer vi ingenting siden de alltid er av typen *int*. Det betyr at algoritmens logikk ikke er endret.

Ulikhetstegnet > gjelder ikke for tegnstrenger. En *maks*-metode for tegnstrenger må isteden bruke metoden *compareTo*. La *s* og *t* være tegnstrenger og la k = s.*compareTo*(t). Da gjelder: Hvis k < 0, kommer *s* foran *t* alfabetisk, hvis *k* er lik 0 er *s* og *t* like og hvis *k* > 0 kommer *s* etter *t* alfabetisk. Det er fortegnet som teller. Om *k* er –1 eller –100 spiller ingen rolle. Det er den negative verdien som forteller at *s* kommer foran *t* alfabetisk. Se *Oppgave* 3 når det gjelder den eksakte returverdien til s.*compareTo*(t). Maks-metoden blir slik:

```
public static int maks(String[] a)    // legges i class Tabell
{
  int m = 0;                           // indeks til største verdi
  String maksverdi = a[0];             // største verdi

  for (int i = 1; i < a.length; i++) if (a[i].compareTo(maksverdi) > 0)
  {
    maksverdi = a[i];  // største verdi oppdateres
    m = i;             // indeks til største verdi oppdaters
  }
  return m;  // returnerer posisjonen til største verdi
}
```

Programkode* 1.4.1 *b)

Programkode 1.4.1 *b)* har nøyaktig samme logikk som *Programkode* 1.4.1 *a)*. De eneste endringene (skrevet med rødt) er at *String* brukes som datatype istedenfor *double* og at sammenligningen a[i] > maksverdi er byttet til a[i].*compareTo*(maksverdi) > 0.

Her er et eksempel på hvordan *maks*-metodene for *int*, *double* og *String* kan brukes (skal det virke må *maks*-metodene ligge i samleklassen Tabell):

```
int[] a = {5,2,7,3,9,1,8,4,6};
double[] d = {5.7,3.14,7.12,3.9,6.5,7.1,7.11};
String[] s = {"Sohil","Per","Thanh","Fatima","Kari","Jasmin"};

int k = Tabell.maks(a);     // posisjonen til den største i a
int l = Tabell.maks(d);     // posisjonen til den største i d
int m = Tabell.maks(s);     // posisjonen til den største i s

System.out.println(a[k] + "  " + d[l] + "  " + s[m]);

// Utskrift:  9  7.12  Thanh
```

Programkode* 1.4.1 *c)

Oppgaver til Avsnitt 1.4.1

1. Legg de to *maks*-metodene fra *Programkode* 1.4.1 *a)* og *b)* inn i samleklassen Tabell og lag et program som utfører *Programkode* 1.4.1 *c)*.
2. Lag en *maks*-metode som finner posisjonen til den «største» verdien i en *char*-tabell. Hvor mange endringer må du gjøre hvis du tar utgangspunkt i *maks*-metoden for datatypen double, dvs. *Programkode* 1.4.1 *a)*. Test metoden din ved å legge inn en *char*-tabell i *Programkode* 1.4.1 *c)*. Hint: En *char*-tabell *c* som for eksempel inneholder tegnene J, A, S, M, I og N, kan lages slik: `char[] c = "JASMIN".toCharArray();`
3. Lag en *maks*-metode som finner posisjonen til den største verdien i en Integer-tabell. Hvor mange endringer må du gjøre hvis du tar utgangspunkt i *maks*-metoden for tegnstrenger, dvs. *Programkode* 1.4.1 *b)*. Test metoden. En testtabell kan du opprette slik: `Integer[] a = {5,2,7,3,9,1,8,4,6};`
4. La *a* og *b* være ta variabler av typen Integer. Finn ut f.eks. ved å eksperimentere, hva `a.compareTo(b)` returnerer. Sett så opp den regelen som metoden er kodet etter.
5. Finn ut, ved å eksperimentere, hva metoden *compareTo* i class String returnerer. Du kan f.eks. lage et program som inneholder:

   ```
   String s = "A", t = "B";
   System.out.println(s.compareTo(t));
   ```

   Se hvilket tall utskriften gir. Bytt så ut *A* og *B* med andre bokstaver, og se om du finner et mønster. Hva blir det hvis *s* = "A" og *t* = "a" ? Hvilket tall gir utskriften hvis *s* = "Æ" og *t* = "Å" ? Hva hvis *s* = "Ø" og *t* = "Å" ? (Se også *Avsnitt* 1.4.10). Bruk så ord istedenfor enkelttegn. Se spesielt på situasjonen der *s* utgjør første del av *t* eller *t* første del av *s*, for eksempel *s* = "Karianne" og *t* = "Kari". Kildekoden til class String vil gi deg fasiten.
6. Ulikhetstegn gjelder ikke for boolean. F.eks. er *false* < *true* ulovlig. Men Boolean har metoden public static int *compare*(boolean x,boolean y) og ved hjelp av den kan vi finne hva som regnes som «minst» og «størst» av *false* og *true*. Hva blir utskriften:

   ```
   System.out.println(Boolean.compare(false, true));
   ```

1.4.2 Sammenlignbarhet, grensesnitt og generiske metoder

Ordet *generisk* (eng: generic, latin: genus = slekt) brukes ofte i programmering. Ordet er mest kjent fra fag som biologi og botanikk. Planter, dyr og andre levende vesener deles opp i arter. En art (eng: species) defineres grovt sett som den maksimale gruppen av individer der individene ligner på hverandre og kan få fruktbare avkom. Generisk brukes om egenskaper. En generisk egenskap er en egenskap som er felles for alle individene innen samme art.

I programmering brukes begrepet generisk både om metoder og klasser. Med tanke på avsnittet over burde en generisk metode virke eller kunne brukes for alle datatyper av samme «art». F.eks. kunne vi si at alle datatyper som er «sammenlignbare» (dvs. at instanser kan sammenlignes innbyrdes og ordnes i en bestemt rekkefølge) utgjør samme «art».

Datatypene *int*, *double* og *String* som vi brukte i *Avsnitt* 1.4.1, er alle sammenlignbare. Tall sammenlignes med hensyn på størrelse, og tegnstrenger med hensyn på alfabetisk orden. De to første (*int* og *double*) hører til de *grunnleggende* datatypene (eng: primitive types), mens *String* derimot er en *referansetype* (eng: reference type). De grunnleggende typene kan sammenlignes vha. ulikhetstegnene (<, <=, >, >=), men det går ikke for referansetypene. De kan kun sammenlignes ved hjep av en metode. Derfor er det ikke mulig å lage en felles *maks*-metode for alle «sammenlignbare» typer. Men vi kan få det til for alle referansetyper T som er «comparable», dvs. er en subtype til grensesnittet `Comparable<T>`.

I eksemplet i *Programkode* 1.4.1 *c)* kan det se ut som at det er den samme metoden som kalles tre ganger, dvs. at metoden dekker både to grunnleggende typer og en referansetype. Men, som vi vet, er det tre forskjellige *maks*-metoder som kalles. Java tillater at samme metodenavn brukes for flere metoder så sant metodene har forskjellige signaturer, dvs. har forskjellige parametere (type og rekkefølge). I *Programkode* 1.4.1 *c)* velger kompilatoren rett metode ut fra datatype (eng: inference). Dermed kunne vi si at *maks*-metoden er generisk selv om det egentlig er snakk om forskjellige metoder med samme navn. I språket C++ er det annerledes. Ved å bruke *template* og *operator overloading* er det der mulig å kode én *maks*-metode og så bruke den for både grunnleggende typer og referansetyper.

I Java er en generisk *maks*-metode en metode som har en *typeparameter*:

```
public static <T> int maks(T[] a)
```

T står for type (eller datatype) og vinkelparentesen <T> forteller at T er en *typeparameter*. Slik som det nå står kan T bety hvilken som helst referansetype (men ikke en grunnleggende type). Men det er kun for «sammenlignbare» referansetyper at en slik *maks*-metode er aktuell. Derfor må typen T ha en metode som sammenligner. Dette ordnes ved hjelp av et *grensesnitt* (eng: interface). I `java.lang` er det generiske grensesnittet `Comparable<T>` definert slik:

```
public interface Comparable<T>   // definert i java.lang
{
  public int compareTo(T o);
}
```

Programkode* 1.4.2 *a)

Dette kalles et *generisk grensesnitt* fordi det inneholder en typeparameter T. Vi kan bruke det til å definere hva det skal bety at instanser av en referansetype kan sammenlignes:

> **Definisjon 1.4.2** *Instanser av referansetypen T kan sammenlignes hvis T implementerer Comparable<T>. Typen T sies da å være sammenlignbar.*

En datatype T som oppfyller Definisjon 1.4.2, vil garantert ha metoden *compareTo*. Vi kan nå informere *maks*-metoden om at typeparameter T ikke er en vilkårlig referansetype, men en der instanser kan sammenlignes, ved å bruke <T extends Comparable<T>> som *begrensning* av typen. Det betyr at T er begrenset til å være en *subtype* til Comparable<T>. Dette vil virke for en T som oppfyller Definisjon 1.4.2 og for en subtype til en slik type. Men det vil være noen spesielle tilfeller som faller utenfor. De kan imidlertid fanges opp ved å bruke: T extends Comparable<? super T> som typeparameter. Se *Oppgave* 6. Dermed kan vi gjøre om *Programkode* 1.4.1 *b)* til flg. generiske *maks*-metode:

```
public static <T extends Comparable<? super T>> int maks(T[] a)
{
  int m = 0;                      // indeks til største verdi
  T maksverdi = a[0];             // største verdi

  for (int i = 1; i < a.length; i++) if (a[i].compareTo(maksverdi) > 0)
  {
    maksverdi = a[i];  // største verdi oppdateres
    m = i;             // indeks til største verdi oppdaters
  }
  return m;  // returnerer posisjonen til største verdi
} // maks
```

Programkode* 1.4.2 *b)

I generiske metoder, klasser og grensesnitt må det inngå én eller flere typeparametere. Da er det ofte gunstig å bruke en stor bokstav som navn. T er et vanlig navn siden det er første bokstav i ordet Type. Hvis det trengs flere, kunne en bruke S eller kanskje U. Men alle bokstaver kan selvfølgelig brukes. Noen eksempler er E for element, V for verdi (eng: value) og K for nøkkelverdi (eng: key). Noen liker også å bruke korte ord med stor forbokstav, f.eks. ordet Type. Sjekk kildekoden til Java og se hvordan det er gjort der.

I *Avsnitt* 1.4.1 benyttet vi at klassen *String* har en *compareTo*-metode. I biblioteket *java.lang* starter klassedefinisjonen av *String* slik:

```
public final class String
  extends Object implements Serializable, Comparable<String>, CharSequence
```

Her står det i klartekst at klassen implementer Comparable<String>. Dermed må den ha en *compareTo*-metode med String som parametertype. I tillegg er klassen konstant (Java: final), dvs. at det ikke er mulig å lage subklasser. Grensesnittet CharSequence inneholder metoder for hente tegn fra tegnstrengen.

Hvis vi legger *maks*-metoden fra *Programkode* 1.4.2 *b)* inn i samleklassen Tabell, vil den inneholde to *maks*-metoder som begge kan brukes for String-tabeller. Det er:

```
public static int maks(String[] a)
public static <T extends Comparable<? super T>> int maks(T[] a)
```

Programkode* 1.4.2 *c)

I programbiten nedenfor kalles en *maks*-metode fra Tabell. Men hvem av dem velges?

```
String[] s = {"Sohil","Per","Thanh","Fatima","Kari","Jasmin"};
int k = Tabell.maks(s);         // hvilken maks-metode?
System.out.println(s[k]);       // Utskrift:  Thanh
```

Programkode* 1.4.2 *d)

Det er den første av de to fra *Programkode 1.4.2 c)* som blir kalt. Regelen er at hvis det finnes en metode med eksakt samme signatur (navn og parametere (type og rekkefølge)) som i kallet, vil den bli kalt. Hvis vi fjerner *maks*-metoden med `String` som paramtertype fra samleklassen `Tabell` , vil programbiten fortsatt virke og gi samme utskrift. Men nå er det den gjenværende *maks*-metoden som blir kalt. Sjekk at det stemmer!

Hvis flere metoder stemmer med hensyn på parametertypen, men ingen eksakt, så er det mer komplisert. Da er det parametertypens plassering i arvehierarkiet som er avgjørende. I verste fall blir det syntaksfeil fordi det oppstår tvetydighet. Se *Oppgave 2 - 4*.

Det ikke mulig å lage én *maks*-metode som virker både for grunnleggende datatyper og for referansetyper. *Programkode 1.4.2 b)* dekker referansetypene. Klassen `Arrays` i `java.util` har f.eks. hele 18 versjoner av sorteringsmetoden *sort*.

Innsettingssorteringen i *Programkode 1.3.8 c)* kan gjøres generisk ved å bytte `temp < a[j]` med `temp.compareTo(a[j]) < 0`:

```
public static <T extends Comparable<? super T>> void innsettingssortering(T[] a)
{
  for (int i = 1; i < a.length; i++)  // starter med i = 1
  {
    T verdi = a[i];         // verdi er et tabellelemnet
    int  j = i - 1;         // j er en indeks
    // sammenligner og forskyver:
    for (; j >= 0 && verdi.compareTo(a[j]) < 0 ; j--) a[j+1] = a[j];

    a[j + 1] = verdi;       // j + 1 er rett sortert plass
  }
}
```

Programkode 1.4.2 e)

Hvis *Programkode* 1.4.2 *e)* er lagt i klassen `Tabell`, vil flg. kodebit være kjørbar:

```
String[] s = {"Per","Kari","Ole","Anne","Ali","Eva"};
Tabell.innsettingssortering(s);
System.out.println(Arrays.toString(s));  // [Ali, Anne, Eva, Kari, Ole, Per]
```

Programkode 1.4.2 f)

Oppgaver til Avsnitt 1.4.2

1. Legg inn *maks*-metoden fra *Programkode 1.4.2 b)* i klassen `Tabell` (legg også inn, hvis du ikke gjorde det i forrige avsnitt, *maks*-metoden fra 1.4.1 *b*). Lag et program som utfører *Programkode 1.4.2 d)*. Hvilken metode velges? Fjern så *maks*-metoden for `String` (dvs. *Programkode* 1.4.1 *b*) fra `Tabell` og utfør *Programkode 1.4.2 d)* på nytt.
2. Legg *Programkode 1.4.2 e)* i klassen `Tabell` og sjekk at *Programkode 1.4.2 f*) virker.
3. `Integer` er subklasse til *Number* som igjen er subklasse til `Object`. Løs denne oppgaven.
4. To ulike metoder kan være like riktige valg for en datatype. Løs denne oppgaven.
5. La en klasse ha metodene `public static void f(int a, float b){}` og `public static void f(float a, int b){}`. De er forskjellige siden parameterlistene er forskjellige. Lag så et program med metodekallet `f(1,1);` Hva sier kompilatoren? Hva skjer hvis den ene *f*-metoden kommenteres vekk? Hva hvis den andre kommenteres vekk?
6. Her skal vi se på forskjellen mellom `<T extends Comparable<T>>` og `<T extends Comparable<? super T>>`. Løs denne oppgaven.

1.4.3 Omslagsklasser

Det vil være upraktisk alltid å måtte kode en algoritme for alle aktuelle grunnleggende typer. Ta f.eks. en *maks*-metode. En helgardering tilsier at vi, i tillegg til den generiske versjonen i *Programkode 1.4.2 b)*, også måtte lage versjoner for `byte`, `short`, `int`, `long`, `char`, `float` og `double`. Heldigvis har Java en teknikk som gjør det litt enklere for oss.

Hver grunnleggende datatype kan «gjøres om» til en referansetype ved hjelp av en såkalt *omslagsklasse* (eng: wrapper class). For eksempel er klassen `Integer` en omslagsklasse for datatypen `int`. Klassen `Integer` har kun et heltall, dvs. en `int`, som instansvariabel. Navnet omslagsklasse kommer av at klassen fungerer som et omslag rundt datatypen. I biblioteket `java.lang` starter klassedefinisjonen av `Integer` slik:

```
public final class Integer extends Number implements Comparable<Integer>
```

Klassen `Integer` har vært med i Java siden starten, men senere har den fått mer innhold. Legg merke til at `Integer` implementerer `Comparable<Integer>`. Det betyr at `Integer` er sammenlignbar. De generiske metodene i *Programkode 1.4.2 b)* og *Programkode 1.4.2 e)* kan dermed brukes på en `Integer`-tabell.

Java har en form for typekonvertering mellom `int` og `Integer`. Det kalles autoboksing (eng: autoboxing) og avboksing (eng: unboxing). En `Integer` kan ses på som et omslag (eller en boks) rundt et heltall. Bruker vi en `int` der syntaksen krever en `Integer`, vil kompilatoren automatisk legge et «omslag» rundt heltallet. Omvendt vil kompilatoren automatisk fjerne «omslaget» hvis vi bruker en `Integer` der syntaksen krever en `int`.

Flg. eksempel viser hvordan autoboksing og avboksing virker:

```
Integer k = new Integer(5);      // gammel (men lovlig) måte
Integer n = 5;                   // ny måte - autoboksing

int i = k.intValue();            // i blir 5 - gammel (men lovlig) måte
int j = n;                       // j blir 5 - avboksing

Integer[] a = {5,2,7,3,9,1,8,10,4,6};   // her inngår autoboksing

Tabell.innsettingssortering(a);  // den generiske metoden kalles
System.out.println(Arrays.toString(a));

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

Programkode 1.4.3 a)

I setningen `Integer[] a = {5,2,7,3,9,1,8,10,4,6}` det ikke en `int`-tabell som automatisk konverteres til en `Integer`-tabell, men hvert heltall i oppramsingen konverteres (autoboksing) til en `Integer`. Hvis vi allerede har en `int`-tabell, må det lages kode for å få en `Integer`-tabell med samme innhold:

```
int[] a = {5,2,7,3,9,1,8,10,4,6};          // en int-tabell
Integer[] b = a;                            // syntaksfeil
```

Programkode 1.4.3 b)

Setningen `Integer[] b = a;` over gir feilmeldingen «`incompatible types: int[] cannot be converted til Integer[]`». Vi må isteden gjøre slik:

```
int[] a = {5,2,7,3,9,1,8,10,4,6};                // en int-tabell

Integer[] b = new Integer[a.length];             // en tom Integer-tabell

for (int i = 0; i < b.length; i++) b[i] = a[i];  // fyller tabellen (autoboksing)
```

Programkode* 1.4.3 *c)

Obs: Det er viktig å være klar over at konstruksjonen i *Programkode* 1.4.3 *c)* er kostbar. Tabellen *b* som har samme innhold, bruker dobbelt så mye plass som *a*.

Java har omslagsklasser for alle de grunnleggende datatypene. Det er `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character` og `Boolean`. Alle er sammenlignbare. Det gjelder til og med `Boolean` der `false` regnes som «mindre enn» `true`. Også de to spesialklassene `BigInteger` og `BigDecimal` er sammenlignbare. Forøvrig har Java mer enn 150 sammenlignbare klasser, dvs. klasser som implementerer grensesnittet `Comparable`.

I forbindelse med effektivitetstesting av generiske metoder kan det være gunstig å kunne arbeide med store tabeller som har et tilfeldig innhold. I *Avsnitt* 1.1.8 ble det utviklet en teknikk for å lage tilfeldige heltallstabeller. Vi kan bruke samme teknikk til å lage tilfeldige `Integer`-tabeller. Legg flg. metoder i samleklassen `Tabell`:

```
public static void bytt(Object[] a, int i, int j)
{
  Object temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}

public static Integer[] randPermInteger(int n)
{
  Integer[] a = new Integer[n];               // en Integer-tabell
  Arrays.setAll(a, i -> i + 1);               // tallene fra 1 til n

  Random r = new Random();   // hentes fra  java.util

  for (int k = n - 1; k > 0; k--)
  {
    int i = r.nextInt(k+1);  // tilfeldig tall fra [0,k]
    bytt(a,k,i);             // bytter om
  }
  return a;  // tabellen med permutasjonen returneres
}
```

Programkode* 1.4.3 *d)

Hvis metodene i *Programkode* 1.4.3 *d)* ligger i klassen `Tabell`, vil flg. kode være kjørbar:

```
Integer[] a = Tabell.randPermInteger(10);
System.out.println(Arrays.toString(a));
// En mulig utskrift: [7, 1, 8, 2, 10, 3, 4, 6, 5, 9]

Tabell.innsettingssortering(a);
System.out.println(Arrays.toString(a));
// Utskrift: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Programkode* 1.4.3 *e)

Oppgaver til Avsnitt 1.4.3

1. Autoboksing og avboksing gjelder ikke bare for `int` og `Integer`, men for alle de grunnleggende typene med tilhørende omslagsklasser. Sjekk dette!
2. Omtrent hver gang Java kommer i ny versjon er det noen nye ting i klassen `Integer`. I API-en vil en se i hvilken versjon det har kommet nye ting. Ta en rask gjennomgang av API-en og se om det er konstanter og metoder der som kan være nyttige. Noen viktige nyheter i Java 8 er metoder for å kunne arbeide uten fortegn. Her er noen eksempler på metoder: `toUnsignedString`, `parseUnsignedInt`, `compareUnsigned`, `remainderUnsigned` og `divideUnsigned`. Hva tror du flg. metodekall returnerer: `Integer.compare(-1, 1);` ? Hva med: `Integer.compareUnsigned(-1, 1);` ?
3. Det er også mange nyheter i de andre omslagsklassene. Gå inn på API-ene og se hva som er nytt.
4. Det er også lovlig å konvertere fra `int` til `Integer` eller fra `Integer` til `int` ved å bruke den vanlige konverteringsteknikken (Java: type casting). Med andre ord er syntaksen i setningene `Integer i = (Integer)10;` og `int j = (int)i;` tillatt. Sjekk at det stemmer! Sjekk også at samme syntaks er tillatt for de andre omslagsklassene.
5. Lag metoden `public static void skriv(Object[] a, int fra, int til)`. Den skal skrive elementene (mellomrom mellom hvert) fra `a[fra:til>` på én linje (uten mellomrom til slutt). Lag så en `skriv`-metode som skriver ut hele *a*. Lag også tilsvarende metoder med navn `skrivln` (de skal avslutte med linjeskift). Lag også metoden `public static void bytt(Object[] a, int i, int j)`. Den skal bytte om elementene på plassene *i* og *j* i tabellen *a*. Legg alle metodene i samleklassen `Tabell`.
6. Legg `bytt` og `randPermInteger` fra *Programkode* 1.4.3 *d)* inn i samleklassen `Tabell` og sjekk at *Programkode* 1.4.3 *e)* virker. Lag større tabeller, f.eks. med 20 verdier.
7. Gitt tabellen: `double[] d = {5.7,3.14,7.12,3.9,6.5,7.1,7.11};` Lag kode som lager en `Double`-tabell ved hjelp av den gitte `double`-tabellen og som så bruker den generiske innsettingssorteringen. Se *Programkode* 1.4.3 *c)*.
8. Det kan oppstå tvetydigheter hvis primitive typer og omslagsklasser blandes. Metoden `public static void f(int a, Integer b) { }` har først en `int` og så en `Integer` som argument. I metoden `public static void f(Integer a, int b) { }` er det motsatt. Dermed er disse metodene forskjellige. Hva vil kompilatoren si til metodekallet `f(1,1);` Hva skjer hvis en av metodene kommenteres vekk? Hvis begge metodene finnes, hvordan kan en da endre i kallet `f(1,1);` for at en bestemt av dem skal brukes?
9. Klassen `BigInteger` ligger i `java.math`. Lag en `BigInteger`-tabell og bruk først den generiske `maks`-metoden til å finne posisjonen til den største verdien og bruk så den generiske innsettingssorteringen til å sortere tabellen. Se også *Avsnitt* 1.7.18.
10. Klassen `BigDecimal` ligger i `java.math`. Lag en `BigDecimal`-tabell og bruk først den generiske `maks`-metoden til å finne posisjonen til den største verdien og bruk så den generiske innsettingssorteringen til å sortere tabellen.

1.4.4 Hvordan implementere et generisk grensesnitt?

Som eksempel lager vi en egen omslagsklasse for datatypen `int` med `Heltall`. Den skal kun inneholde de mest nødvendige metodene. Legg den under *package eksempelklasser*:

```
public final class Heltall implements Comparable<Heltall>
{
  private final int verdi;    // et heltall som instansvariabel

  public Heltall(int verdi) { this.verdi = verdi; }   // konstruktør

  public int intVerdi() { return verdi; }             // aksessor

  public int compareTo(Heltall h)        // Heltall som parameter
  {
    return verdi < h.verdi ? -1 : (verdi == h.verdi ? 0 : 1);
  }

  public boolean equals(Object o)
  {
    if (o == this) return true;   // sammenligner med seg selv
    if (!(o instanceof Heltall)) return false;  // feil datatype
    return verdi == ((Heltall)o).verdi;
  }

  public boolean equals(Heltall h) { return verdi == h.verdi; }

  public int hashCode() { return 31 + verdi; }

  public String toString() { return Integer.toString(verdi); }

} // class Heltall
```

Programkode* 1.4.4 *a)

Klassen skal implementere `Comparable<Heltall>` og må da ha en *compareTo*-metode. Der er den trinære operatoren ?: brukt to ganger og det gir kort og effektiv kode. Men det kan lett erstattes med en `if - else if - else` slik som dette (se også *Oppgave* 1):

```
if (verdi < h.verdi) return -1;
else if (verdi == h.verdi) return 0;
else return 1;
```

Programkode* 1.4.4 *b)

Returverdiene til *compareTo*-metoden skal gjenspeile den vanlige tallordningen. Da er det mye å passe på. Hvis f.eks. $x < y$, så er selvfølgelig $y > x$. Det betyr at metoden må kodes slik at hvis `x.compareTo(y) < 0`, så må `y.compareTo(x) > 0`. Dette og andre krav er oppfylt slik *compareTo*-metoden er kodet her. De formelle kravene til hvordan en *compareTo*-metode må kodes tas opp i *Avsnitt* 1.4.12.

Alle klasser arver metoden *equals* fra basisklassen `Object`. To instanser x og y sies å være like hvis `x.equals(y) == true`. I heltallsklassen kan også *compareTo*-metoden brukes til å avgjøre likhet, dvs. x og y er like hvis `x.compareTo(y) == 0`. Her må vi ha konsistens, dvs. at `x.equals(y) == true` er ekvivalent med `x.compareTo(y) == 0`. Den arvede *equals*-metoden må overskrives. Det er gjort i *Programkode* 1.4.4 *a)*. Mer om dette i *Avsnitt* 1.4.12.

Metodene *toString* og *hashCode* (som arves fra Object) bør overskrives siden de brukes implisitt. Spesielt må *hashCode* være konsistent med *equals*. Mer om det senere.

Heltall kan mange steder erstatte Integer. Men det er ingen autoboksing og avboksing mellom int og Heltall slik som det er mellom int og Integer (se *Avsnitt* 1.4.3). Derfor må vi bruke tradisjonelle teknikker for overgangen mellom int og Heltall. Se flg. eksempel:

```
int[] a = {5,2,7,3,9,1,8,10,4,6};          // en int-tabell
Heltall[] h = new Heltall[a.length];       // en Heltall-tabell

for (int i = 0; i < h.length; i++) h[i] = new Heltall(a[i]);
Tabell.innsettingssortering(h);           // generisk sortering
System.out.println(Arrays.toString(h));   // utskrift
```

Programkode* 1.4.4 *c)

Et eksempel til: Klassen Person er for personopplysninger. Den gjøres sammenlignbar ved å ordne personer alfabetisk, dvs. etternavn først og så etter fornavn hvis etternavnene er like. Vi antar at alle har forskjellige navn. Legg den under *package eksempelklasser*:

```
public class Person implements Comparable<Person>
{
  private final String fornavn;         // personens fornavn
  private final String etternavn;       // personens etternavn

  public Person(String fornavn, String etternavn)   // konstruktør
  {
    this.fornavn = fornavn;
    this.etternavn = etternavn;
  }

  public String fornavn() { return fornavn; }       // aksessor
  public String etternavn() { return etternavn; }   // aksessor

  public int compareTo(Person p)    // pga. Comparable<Person>
  {
    int cmp = etternavn.compareTo(p.etternavn);     // etternavn
    if (cmp != 0) return cmp;             // er etternavnene ulike?
    return fornavn.compareTo(p.fornavn);  // sammenligner fornavn
  }

  public boolean equals(Object o)      // vår versjon av equals
  {
    if (o == this) return true;
    if (!(o instanceof Person)) return false;
    return compareTo((Person)o) == 0;
  }

  public int hashCode() { return Objects.hash(etternavn, fornavn); }

  public String toString() { return fornavn + " " + etternavn; }

} // class Person
```

Programkode* 1.4.4 *d)

I *compareTo*-metoden sammenlignes først etternavnene. Det skjer ved hjelp av *compareTo* for `String` siden `etternavn` er en instans av den klassen. Resultatet legges i hjelpevariablen *cmp*. Hvis *cmp* != 0, har personene forskjellige etternavn. Vi får da rett fortegn hvis *cmp* returneres. Hvis etternavnene er like (dvs. *cmp* == 0), sammenlignes fornavnene og det skjer også ved hjelp av *compareTo* for `String`.

Den ordningen vi har innført for personer kalles en *leksikografisk* ordning. Det betyr at ordningen er leddvis og at hvert ledd har sin egen ordning. Her er det etternavn og fornavn som utgjør de to leddene. Mer om leksikografiske ordninger i *Avsnitt* 1.4.8.

Vi kan nå sortere personer etter den gitte ordningen ved hjelp av den den generiske metoden fra *Programkode* 1.4.2 *e)*. I flg. eksempel inngår de fem personene Kari Svendsen, Boris Zukanovic, Ali Kahn, Azra Zukanovic og Kari Pettersen. Først buker vi den generiske *maks*-metoden fra *Programkode* 1.4.2 *b)* til å finne den «største» personen, dvs. den som kommer sist i den leksikografiske ordningen. Denne personen skrives ut. Så sorteres tabellen ved hjelp av innsettingssorteringen og resultatet skrives ut:

```
Person[] p = new Person[5];                    // en persontabell

p[0] = new Person("Kari","Svendsen");          // Kari Svendsen
p[1] = new Person("Boris","Zukanovic");        // Boris Zukanovic
p[2] = new Person("Ali","Kahn");               // Ali Kahn
p[3] = new Person("Azra","Zukanovic");         // Azra Zukanovic
p[4] = new Person("Kari","Pettersen");         // Kari Pettersen

int m = Tabell.maks(p);                        // posisjonen til den største
System.out.println(p[m] + " er størst");       // skriver ut den største

Tabell.innsettingssortering(p);                // generisk sortering
System.out.println(Arrays.toString(p));        // skriver ut sortert

// Utskrift:

// Boris Zukanovic er størst
// [Ali Kahn, Kari Pettersen, Kari Svendsen, Azra Zukanovic, Boris Zukanovic]
```

Programkode* 1.4.4 *e)

Vi kunne også bruke en ferdig sorteringsmetode fra klassen `Arrays` i biblioteket `java.util`. Bytt ut den første setningen i *Programkode* 1.4.4 *g)* med `Arrays.sort(p)`. Da vil vi få samme resultat. Prøv dette!

Java har ingen ferdig generisk metode som finner posisjonen til den største verdien i en tabell av sammenlignbare verdier, f.eks. en `Person`-tabell. Men Java har en teknikk for å finne den største verdien, men da via en liten omvei. Da må vi midlertid ta i bruk ting som kun delvis er forklart tidligere. Noe av det blir forklart senere. Se hva resultatet blir når følgende kode legges til slutt i *Programkode* 1.4.4 *e)* og hele programmet kjøres (obs: du må øverst ha med `import java.util.stream.*;`):

```
Stream s = Arrays.stream(p);
Optional<Person> resultat = s.max(Comparator.naturalOrder());
resultat.ifPresent(System.out::println);
```

Programkode* 1.4.4 *f)

Oppgaver til Avsnitt 1.4.4

1. Kopier `class Heltall` fra *Programkode 1.4.4 a)* over til deg selv. Opprett mappen (package) *eksempelklasser* og legg den der.

   a) Lag og kjør et program med *Programkode 1.4.4 c)*.

   b) Bruk *Programkode 1.4.4 b)* i *compareTo*-metoden. Sjekk at alt virker som før!

   c) La `return verdi - h.verdi;` være kode i *compareTo*-metoden. Matematisk sett blir det korrekt. Sjekk at alt virker som før. Det er imidlertid et problem her. Hva er det?

   d)
   ```
   Heltall x = new Heltall(3), y = new Heltall(3);  // x og y er like
   System.out.println(x.compareTo(y) + "  " + x.equals(y));
   ```

   Lag et program som inneholder koden over. Hva blir utskriften? Kommenter så vekk *equals*-metoden i `class Heltall`. Da vil det bli den versjonen av `equals` som arves fra `class Object`, som brukes. Hva blir nå utskriften? Slå opp i kildekoden til `class Object` og se hvordan metoden *equals* er kodet der.

   e)
   ```
   Heltall x = new Heltall(3), y = new Heltall(3);  // x og y er like
   System.out.println(x.hashCode() + "  " + y.hashCode());
   ```

   Lag et program som inneholder koden over. Hva blir utskriften? Kommenter så vekk *hashCode*-metoden i `class Heltall`. Da vil det bli den versjonen av *hashCode* som arves fra `class Object`, som brukes. Hva blir nå utskriften? Slå opp i kildekoden til `class Object` og se hva som står der om metoden *hashCode*.

2. Kopier klassen `Person` fra *Programkode 1.4.4 d)* over til deg. Legg den under *package eksempelklasser*:

   a) Legg inn flere personer i `Person`-tabellen i *Programkode 1.4.4 e)*.

   b) Kjør *Programkode 1.4.4 e)* etter at du har gjort som i a).

   c) Bruk `sort`-metoden fra `class Arrays` i *Programkode 1.4.4 e)*.

   d) Legg inn kode i konstruktøren slik at det kastes en `NullPointerException` hvis fornavn eller etternavn er null. Ta med en tekst som forteller hvilket navn som er null.

   e) Lag en mer direkte versjon av metoden *equals*(), dvs. uten å bruke *compareTo*(). Bruk også `getClass() != o.getClass()` istedenfor `!(o instanceof Person)`. Men da må det først være kode som returnerer *false* hvis *o* er `null`. Hvis ikke, vil `o.getClass()` kaste en `NullPointerException`.

   f) Lag metoden `public boolean` *equals*`(Person p)`. Da trengs ingen typesjekking.

   g) Metoden *hashCode*() i klassen `Person` kan kodes på mange måter. Hvis *equals*() er kodet, men ikke *hashCode*(), vil f.eks. *NetBeans* og *Eclipse* si ifra og samtidig komme med forslag om hvordan *hashCode*() skal kodes. Sjekk dette. Obs: I *NetBeans* er dette en del av standardoppsettet. I *Eclipse* må du selv sette det som en opsjon. Men du får tilbudet i menyvalget *Source*. Hash-teknikk blir tatt opp mer grundig i *Kapittel 6*.

   h) I metoden *toString* i klassen *Person* skjøtes fornavn, et mellomrom og etternavn sammen. Dette kan også gjøres ved hjelp av metoden `join` i klassen `String`. Prøv på det!

   i) Legg *Programkode 1.4.4 f)* bakerst i *Programkode 1.4.4 e)*. Kjør programmet!

   j) *Programkode 1.4.4 f)* inneholder tre programlinjer. Gjør det om slik at det blir kun én programlinje.

1.4.5 Subtyper til sammenlignbare typer

Typebegrensningen `<T extends Comparable<? super T>>` av typen `T` (eng: bounded type parameter) vil, som nevnt tidligere, gjøre at alle typer `T` som oppfyller Definisjon 1.4.2 og *subtyper* av slike typer, kan brukes. Vi skal sjekke dette ved å lage en subtype `Student` til `Person`. Ved Høgskolen i Oslo og Akershus hører en datastudent til et bestemt studium eller er eventuelt en enkeltemnestudent. De tre datastudiene omtales normalt som Data, IT og Anvendt. Her velger vi å opprette en *enumklasse* `Studium` for de aktuelle studiene:

```
public enum Studium  // legges under package eksempelklasser
{
  Data ("Ingeniørfag - data"),          // enumkonstanten Data
  IT ("Informasjonsteknologi"),         // enumkonstanten IT
  Anvendt ("Anvendt datateknologi"),    // enumkonstanten Anvendt
  Enkeltemne ("Enkeltemnestudent");     // enumkonstanten Enkeltemne

  private final String fulltnavn;       // instansvariabel
  private Studium(String fulltnavn) { this.fulltnavn = fulltnavn; }

  public String toString() { return fulltnavn; }
}
```

Programkode 1.4.5 a)

Flg. kodebit viser hvordan vi kan skrive ut enumtypens konstanter:

```
for (Studium s : Studium.values())
{
  System.out.println(s.toString() + " (" + s.name() + ")");
}
// Ingeniørfag - data (Data)
// Informasjonsteknologi (IT)
// Anvendt datateknologi (Anvendt)
// Enkeltemnestudent (Enkeltemne)
```

Programkode 1.4.5 b)

En student hører som nevnt til et studium. Det kan nå oppgis ved hjelp av en enumkonstant fra `Studium`. En student har selvfølgelig også for- og etternavn og er dermed en person. Vi lager derfor Student som en subklasse til `Person` (legges i *package eksempelklasser*):

```
public class Student extends Person   // Student blir subklasse til Person
{
  private final Studium studium;      // studentens studium

  public Student(String fornavn, String etternavn, Studium studium)
  {
    super(fornavn, etternavn);
    this.studium = studium;
  }

  public String toString() { return super.toString() + " " + studium.name();}

  public Studium studium() { return studium; }

}  // class Student
```

Programkode 1.4.5 c)

Vi utnevner personene i eksemplet fra *Programkode* 1.4.4 *e)* til studenter og tilordner hver av dem et studium. En `Student` er en `Person` og instanser av `Student` kan sammenlignes og ordnes som personer. Derfor kan vi her benytte den generiske innsettingssorteringen fra *Programkode* 1.4.2 *e)* på en `Student`-tabell:

```
Student[] s = new Student[5];  // en Studenttabell

s[0] = new Student("Kari", "Svendsen", Studium.Data);    // Kari Svendsen
s[1] = new Student("Boris", "Zukanovic", Studium.IT);    // Boris Zukanovic
s[2] = new Student("Ali", "Kahn", Studium.Anvendt);      // Ali Kahn
s[3] = new Student("Azra", "Zukanovic", Studium.IT);     // Azra Zukanovic
s[4] = new Student("Kari", "Pettersen", Studium.Data);   // Kari Pettersen

Tabell.innsettingssortering(s);                    // Programkode 1.4.2 e)
for (Student t : s) System.out.println(t);

// Utskrift:
// Ali Kahn Anvendt
// Kari Pettersen Data
// Kari Svendsen Data
// Azra Zukanovic IT
// Boris Zukanovic IT
```

Programkode* 1.4.5 *d)

Det er også mulig (men kanskje ikke spesielt naturlig) å blande personer og studenter i samme persontabell. Det går bra siden en student er en person. De generiske metodene vil virke som tidigere, f.eks. *innsettingssortering*():

```
Person[] p = new Person[5];      // en Persontabell

p[0] = new Student("Kari","Svendsen",Studium.Data);    // en student
p[1] = new Person("Boris","Zukanovic");                // en person
p[2] = new Student("Ali","Kahn",Studium.Anvendt);      // en student
p[3] = new Person("Azra","Zukanovic");                 // en person
p[4] = new Student("Kari","Pettersen",Studium.Data);   // en student

Tabell.innsettingssortering(p);                    // Programkode 1.4.2 e)
for (Person t : p) System.out.println(t);

// Utskrift:
// Ali Kahn Anvendt
// Kari Pettersen Data
// Kari Svendsen Data
// Azra Zukanovic
// Boris Zukanovic
```

Programkode* 1.4.5 *e)

De generiske metodene våre har typebegrensningen `<T extends Comparable<? super T>>`. I de eksemplene der metodene er brukt, ville det ha fungert ok hvis typebegrensningen isteden var `<T extends Comparable<T>>`. Sjekk at det stemmer! Slik var det imidlertid ikke før. Men fra og med Java 8 går det bra. Men vi bruker likevel den første typebegrensningen fordi det er spesielle tilfeller der den trengs. Begrensningen `<? super T>` kalles en *jokerbegrensning* (eng: bounded wildcard). Det betyr at hvis det finnes en supertype `S` til `T` slik at `T` er en subtype til `Comparable<S>`, så vil `T` «slippe gjennom» typebegrensningen. Se *Oppgave* 6 og 7.

Oppgaver til Avsnitt 1.4.5

1. Flytt enumtypen `Studium` over til deg. Legg den under (package) `eksempelklasser`. Lag så et program der *Programkode* 1.4.5 *b)* inngår og kjør programmet.
2. Hvis du ikke allerede er kjent med *enumtyper*, burde du lære deg mer om dem. En enumtype er en referansetype og fungerer omtrent som andre referansetyper (klasser og grensesnitt). Det som ramses opp i typedefinisjonen kalles *enumkonstanter*. De bør normalt ha store bokstaver eller minst ha stor forbokstav. Det er mulig å la navnet starte med _ (understrek), $ (dollar) eller £ (pund), men det brukes svært sjelden. En enumtype er en subtype til `Object` og har dermed alle `Object`-metodene. En av dem er *`toString`*`()` som er overskrevet (eng: overridden) i `Studium`. Hvis ikke, ville *`toString`*`()` og *`name`*`()` gi samme resultat, dvs. navnet på enumkonstanten. Det er ikke mulig å opprette instanser av en enumtype ved hjelp av **`new`**. Trengs det flere, må de settes opp sammen med de andre i «oppramsingen». En enumtype er alltid `Comparable` (og har dermed metoden *`compareTo`*). Ordningen er den rekkefølgen de har i «oppramsingen». Metoden *`ordinal`*`()` forteller hvilket nummer i rekkefølgen en enumkonstant har. Den første har ordinalverdi lik 0, den neste 1, osv.
3. Lag enumtypen `Måned`. Den skal inneholde konstanter for hver måned (med tre bokstaver), `JAN`, `FEB`, . . . , `NOV`, `DES`. La typen få en instansvariabel med navn *`mndnr`* og en aksessmetode med samme navn. Det betyr at `Måned.JAN.`*`mndnr`*`()` skal gi 1, osv. til at `Måned.DES.`*`mndnr`*`()` skal gi 12. La også typen få en instansvariabel med navn *`fulltnavn`*. For JAN skal det være januar, osv. Overskriv *`toString`*`()` slik at den gir det fulle navnet. Lag også `public static String` *`toString`*`(int mnd)` slik at den gir januar for mnd lik 1, osv. La videre `Måned` ha fire statiske metoder *`vår`*`()`, *`sommer`*`()`, *`høst`*`()` og *`vinter`*`()`. Hver av dem skal reture en `Måned`-tabell som inneholder de tilsvarende enumkonstantene. Vi sier at `APRIL` og `MAI` er vår, `JUN`, `JUL` og `AUG` er sommer, `SEP` og `OKT` er høst og at `NOV`, `DES`, `JAN`, `FEB` og `MAR` er vinter. Skriv så ut innholdet ved hjelp av et program av samme type som i *Programkode* 1.4.5 *b)*. Legg `Måned` under package eksempelklasser.
4. Det hender at en elektrostudent tar emnet Algoritmer og datastrukturer. Utvid enumtypen `Studium` slik at det også inngår en enumkonstant `Elektro`. Legge den f.eks. som nr. fire (foran `Enkeltemne`). Studiets navn er: Ingeniørfag - elektronikk og informasjonsteknologi.
5. Flytt klassen `Student` over til deg (package eksempelklasser) og sjekk at *Programkode* 1.4.5 *d)* virker som de skal. Legg inn et par elektrostudenter (se Oppgave 4) og et par enkeltemnestudenter i tabellen i *Programkode* 1.4.5 *d)*. Kjør programmet.
6. Ta vekk `? super` i typebegrensningen `<T extends Comparable<? super T>>`, dvs. slik at det blir `<T extends Comparable<T>>`, i den generiske `maks`-metoden og i den generiske innsettingssorteringen. Sjekk at *Programkode* 1.4.5 *b)* og 1.4.5 *c)* da virker som før.
7. Klassen `Person` starter slik: `public class Person implements Comparable<Person>`. Bruk nå `Comparable<Object>` istedenfor `Comparable<Person>`. Da må signaturen endres fra `compareTo(Person p)` til `compareTo(Object o)`. Det krever videre at det legges inn en ekstra første setning i `compareTo`: `Person p = (Person)o;` Anta så at du har gjort som i *Oppgave* 6. Hva skjer da hvis du vil kjøre 1.4.5 *b)* eller 1.4.5 *c)*? Bring så metodene tilbake til slik de var, dvs. før du gjorde som i *Oppgave* 6. Men behold endringene i klassen `Person`. Hva skjer da hvis du vil kjøre 1.4.5 *b)* eller 1.4.5 *c)*? Konklusjonen blir at vi beholder typebegrensningen `<T extends Comparable<? super T>>`. Den dekker alle aktuelle tilfeller.
8. Du finner svar på nærmest ethvert problem og spørsmål du måtte ha om generiske teknikker i Java ved å gå til Angelika Langers side *Java Generics FAQ*.

1.4.6 Grensesnittet Comparator

Typen Person ordnes med hensyn på etternavn først og så med hensyn på fornavn. Slik er det f.eks. i telefonkatalogen. Men det kan være aktuelt å ordne annerledes. I en forening eller på en arbeidsplass blir man ofte ordnet etter fornavn. Vi kan omkode *compareTo*-metoden i Person. Men da får vi et problem hvis vi ønsker å ordne med hensyn på etternavn et sted og med hensyn på fornavn et annet sted. En klasse kan bare ha én *compareTo*-metode.

Løsningen er å bruke en Komparator. Java har allerede grensesnittet Comparator , men det har en litt komplisert oppbygging. Vi skal derfor lage vår egen enkle versjon først. Det vil gi oss den nødvendige forståelsen for hvordan dette fungerer. Deretter går vi over til å bruke det som Java tibyr. Vi bruker navnet Komparator på vårt eget grensesnitt. Det er for å unngå navnekonflikt med Comparator (legg Komparator under package *eksempelklasser*):

```
@FunctionalInterface                  // legges i mappen eksempelklasser
public interface Komparator<T>        // et funksjonsgrensesnitt
{
  int compare(T x, T y);              // en abstrakt metode
}
```

Programkode 1.4.6 a)

Komparator kalles et *funksjonsgrensesnitt* (eng: functional interface) siden det er nøyaktig én abstrakt metode. Metoden *compare* i Komparator er per definisjon offentlig (public) selv om det ikke står det eksplisitt. Metodens returverdier skal oppfylle de samme kravene som *compareTo*-metoden i grensesnittet Comparable. Det betyr at *compare*(x,y) skal returnere et negativt helttall hvis *x* er «mindre enn» *y* og et positivt heltall hvis *x* er «større enn» *y*. Men den kan returnere 0 uten at *x* og *y* nødvendigvis er like.

Skal vi bruke en komparator til å sortere personer etter fornavn istedenfor etter etternavn, må vi lage en sorteringsmetode som benytter en komparator, dvs. en *komparatormetode*. Det er nok å gjøre noen små endringer i metoden i *Programkode 1.4.2 e)* (den som sorterer instanser av typen Comparable). Vi får da flg. generiske *komparatormetode* - endringene er markert med rødt. Legg den i samleklassen Tabell.

```
public static <T> void innsettingssortering(T[] a, Komparator<? super T> c)
{
  for (int i = 1; i < a.length; i++)  // starter med i = 1
  {
    T verdi = a[i];          // verdi er et tabellelemnet
    int  j = i - 1;          // j er en indeks

    // sammenligner og forskyver:
    for (; j >= 0 && c.compare(verdi,a[j]) < 0 ; j--) a[j+1] = a[j];

    a[j + 1] = verdi;        // j + 1 er rett sortert plass
  }
}
```

Programkode 1.4.6 b)

«Gammeldags» teknikk: Ta utgangspunkt i tabellen i *Programkode 1.4.4 e)*. Den skal nå sorteres mhp. fornavn. Rekkefølgen på personer med samme fornavn er likegyldig. Vi trenger en komparator. Den «gammeldagse» teknikken går ut på (eller gikk ut på) å lage en klasse som implementerer Komparator. La den f.eks. hete FornavnKomparator. Så opprettes en instans av den og den inngår som argument i sorteringsmetoden:

```
public static void main(String... args)
{
  Person[] p = new Person[5];                       // en persontabell
  p[0] = new Person("Kari", "Svendsen");            // Kari Svendsen
  p[1] = new Person("Boris", "Zukanovic");          // Boris Zukanovic
  p[2] = new Person("Ali", "Kahn");                 // Ali Kahn
  p[3] = new Person("Azra", "Zukanovic");           // Azra Zukanovic
  p[4] = new Person("Kari", "Pettersen");           // Kari Pettersen

  class FornavnKomparator implements Komparator<Person>
  {
    public int compare(Person p1, Person p2)        // to personer
    {
      return p1.fornavn().compareTo(p2.fornavn());  // sammenligner fornavn
    }
  }

  Komparator<Person> c = new FornavnKomparator();   // en instans av klassen
  Tabell.innsettingssortering(p, c);                // se Programkode 1.4.6 b)

  System.out.println(Arrays.toString(p));           // Utskrift av tabellen p
  // [Ali Kahn, Azra Zukanovic, Boris Zukanovic, Kari Svendsen, Kari Pettersen]
}
```

Programkode 1.4.6 c)

Argumentene p1 og p2 i fornavnkomparatorens *compare*-metode er personer. Aksessmetoden *fornavn*() gir oss fornavnene og siden de er tegnstrenger, kan de sammenlignes ved hjelp av *compareTo*-metoden, dvs. ved setningen: p1.*fornavn*().*compareTo*(p2.*fornavn*()).

Moderne teknikk: Et *lamda-uttrykk* lager implisitt en instans av en (anonym) klasse som implementerer Komparator. Denne nye Javakonstruksjonen fører til vesentlig kortere kode:

```
Komparator<Person> c = (p1,p2) -> p1.fornavn().compareTo(p2.fornavn());
```

Programkode 1.4.6 d)

Dermed kan koden som kommer etter Person-tabellen i *Programkode* 1.4.6 *c)*, kortes ned til:

```
Komparator<Person> c = (p1,p2) -> p1.fornavn().compareTo(p2.fornavn());
Tabell.innsettingssortering(p, c);                // se Programkode 1.4.6 b)
System.out.println(Arrays.toString(p));           // Utskrift av tabellen p
```

Programkode 1.4.6 e)

Programkode 1.4.6 *e)* kan kortes ned enda mer. Variabelen *c* er egentlig unødvendig. Vi kan isteden la lambda-uttrykket gå direkte inn som argument i metoden. Det er normalt å gjøre det når lamda-utrykket ikke er for stort og komplisert.

```
Tabell.innsettingssortering(p, (p1,p2) -> p1.fornavn().compareTo(p2.fornavn()));
System.out.println(Arrays.toString(p));
```

Programkode 1.4.6 f)

Student er subtype til Person. En student hører til et studium. Vi kan sortere studenter etter studium vha. metoden i *Programkode* 1.4.6 *b)*. Vi bruker et lambda-uttrykk på samme måte som over. Der brukes den naturlige ordningen for enumtypen Studium (dvs. Data, IT, . . .):

```
Student[] s = new Student[5];                              // en studenttabell
s[0] = new Student("Kari","Svendsen", Studium.Data);       // Kari Svendsen
s[1] = new Student("Boris","Zukanovic", Studium.IT);       // Boris Zukanovic
s[2] = new Student("Ali","Kahn", Studium.Anvendt);         // Ali Kahn
s[3] = new Student("Azra","Zukanovic", Studium.IT);        // Azra Zukanovic
s[4] = new Student("Kari","Pettersen", Studium.Data);      // Kari Pettersen

Tabell.innsettingssortering(s, (s1,s2) -> s1.studium().compareTo(s2.studium()));
System.out.println(Arrays.toString(s));
```

Programkode* 1.4.6 *g)

Komparatorene gitt ved lamda-uttrykkene i *Programkode* 1.4.6 *f)* og *Programkode* 1.4.6 *g)* representerer *preordninger* - se *Avsnitt* 1.4.11. Det betyr at metoden *compare* returnerer 0 uten at de to objektene som sammenlignes nødvendigvis er like. To forskjellige personer kan ha samme fornavn. Ordningen vil dermed ikke gi noen bestemt rekkefølge mellom dem.

Vi kan, ved å utvide lambda-uttrykket i *Programkode* 1.4.6 *g)*, ordne studenter på samme studium på en bestemt måte. F.eks. som personer, dvs. først etternavn og så fornavn. For lesbarhetens skyld setter vi opp lamda-uttrykket separat, men det kunne ha gått inn direkte som et argument i sorteringsmetoden. Se *Oppgave* 2. La følgende kodebit erstatte de to siste programsetningene i *Programkode* 1.4.6 *g)*. Kjør så programmet!

```
Komparator<Student> c = (s1,s2) ->
{
  int cmp = s1.studium().compareTo(s2.studium());
  return cmp != 0 ? cmp : s1.compareTo(s2);
};

Tabell.innsettingssortering(s, c);    // Programkode 1.4.6 b)
System.out.println(Arrays.toString(s));
```

Programkode* 1.4.6 *h)

Vi lager en komparator når en datatype som allerede er sammenlignbar (har *compareTo*), skal ordnes på en spesiell måte (forskjellig fra den naturlige ordningen), eller når vi ønsker å ordne en datatype som ikke har noen naturlig ordning, på en eller annen måte. Datatypene `Person` og `Student` har naturlige ordninger (etternavn - fornavn). Det å ordne personer etter fornavn og studenter etter studietilhørighet, må derfor kalles spesielle ordninger.

Også vanlige datatyper som `String` og `Integer`, kan ordnes på spesielle måter. F.eks. kan tegnstrenger ordnes etter lengde, dvs. den av to tegnstrenger som har færrest tegn, skal komme først av de to. Til å avgjøre det bruker vi rett og slett differensen mellom lengdene eller egentlig fortegnet til `x.length() - y.length()`:

```
String[] s = {"Lars","Anders","Bodil","Kari","Per","Berit"};
Tabell.innsettingssortering(s, (s1,s2) -> s1.length() - s2.length());

System.out.println(Arrays.toString(s));
// Utskrift: [Per, Lars, Kari, Bodil, Berit, Anders]
```

Programkode* 1.4.6 *i)

Like lange tegnstrenger får ingen innbyrdes ordning. Hvis vi f.eks. ønsker at de skal ordnes alfabetisk, får vi til det ved å utvide lambda-uttrykket i *Programkode* 1.4.6 *i)*. Se *Oppgave* 5.

Naturlig ordning av `Integer` er etter tallstørrelse. Vi skal nå ordne leksikografisk, dvs. vi ser på sifrene som tegn. Da kommer "1" foran "10" og "10" kommer foran "2", osv.

```
Integer[] a = {13,25,11,3,2,21,10,1,33,100};  // en Integer-tabell
Tabell.innsettingssortering(a, (x,y) -> x.toString().compareTo(y.toString()));

System.out.println(Arrays.toString(a));
// Utskrift: [1, 10, 100, 11, 13, 2, 21, 25, 3, 33]
```

Programkode 1.4.6 j)

I flg. ordning skal ethvert oddetall komme foran ethvert partall. La *x* og y være to heltall. Hvis *x – y* er et partall, så er enten begge oddetall eller begge partall:

```
Komparator<Integer> c = (x,y) ->
{
  if (((x - y) & 1) == 0) return 0;  // x og y oddetall eller x og y partall
  else if ((x & 1) == 0) return 1;   // x partall og y oddetall
  else return -1;                    // x oddetall og y partall
};

Integer[] b = {6,2,7,1,9,5,10,8,4,3};
Tabell.innsettingssortering(b, c);

System.out.println(Arrays.toString(b));
// Utskrift: [7, 1, 9, 5, 3, 6, 2, 10, 8, 4]
```

Programkode 1.4.6 k)

Oddetallene og partallene får ingen bestemt ordning. Hvis vi f.eks. ønsker at de skal ordnes stigende, får vi til det ved å utvide lambda-uttrykket i *Programkode* 1.4.6 *k)*. Se *Oppgave* 6.

Oppgaver til Avsnitt 1.4.6

1. Legg `Komparator` over til deg (under mappen `eksempelklasser`), legg metoden i *Programkode* 1.4.6 *b)* inn i samleklassen `Tabell` og sjekk at *Programkode* 1.4.6 *c)* virker. Gjør så slik som i *Programkode* 1.4.6 *e)* og til slutt slik som i *Programkode* 1.4.6 *f)*.
2. Sjekk at *Programkode* 1.4.6 *g)* virker. Utvid tabellen, dvs. med flere studenter og studier. Erstatt så de to siste linjene med *Programkode* 1.4.6 *h)*. Dropp så komparatorvariabelen *c* og la isteden lambda-uttrykket inngå direkte som argument i sorteringsmetoden.
3. Lag et lamda-uttrykk som ordner studentene etter studium, fornavn og etternavn.
4. Lag metoden `public static <T> int maks(T[] a, Komparator<? super T> c)`. Bruk f.eks. `maks`-metoden som utgangspunkt. Legg den i samleklassen `Tabell`.
5. Utvid lambda-uttrykket i *Programkode* 1.4.6 *i)* slik at like lange strenger ordnes alfabetisk. La så tabellen *s* inneholde "21","18","8","13","20","6","16","25","3","10". Kan du forutsi hvordan den vil bli sortert? Kjør så koden.
6. Utvid lambda-uttrykket i *Programkode* 1.4.6 *k)* slik at oddetallene ordnes internt stigende og partallene internt stigende.
7. Lambda-uttrykket i *Programkode* 1.4.6 *h)* ordner studiene «naturlig» (dvs. rekkefølgen de har i enumen). Gjør om slik at de ordnes alfabetisk etter navn (Anvendt, Data, IT).
8. En `sort`-metode i `class Arrays` bruker en `Comparator`. Der kan lambda-uttrykk, helt maken til de vi har laget over, gå direkte inn som argument i metoden. Sjekk det ved at du bytter ut `Tabell.innsettingssortering` med `Arrays.sort` i *Programkode* 1.4.6 *g)*.

1.4.7 Komparatorer for naturlige ordninger

Sorteringsmetoden i *Programkode* 1.4.2 *e)* virker for datatyper som kan sammenlignes, dvs. for datatyper som oppfyller *Definisjon* 1.4.2 og subtyper av slike, f.eks. `Integer`, `String`, `Person` og `Student`. Den andre sorteringsmetoden, dvs. den i *Programkode* 1.4.6 *b)*, virker for alle typer så sant vi har en komparator (et lambda-uttrykk) for den ordningen vi ønsker, f.eks. det å ordne studenter etter klasse.

En vanlig forutsetning for å sette `T implements Comparable<T>` på en datatype `T` er at instanser av den har en såkalt naturlig ordning. Det gjelder `Integer` som ordnes etter tallstørrelse, `String` som ordnes alfabetisk og `Person` som ordnes slik som i telefonkatalogen.

Vi trenger egentlig ikke sorteringsmetoden i *Programkode* 1.4.2 *e)*. Hvis datatypen har en `compareTo`-metode, kan vi enkelt bruke den til konstruere et lambda-uttrykk:

```
Integer[] a = {6,2,10,9,5,1,8,4,3,7};                          // Integer
String[] s = {"Sohil","Per","Thanh","Ann","Kari","Jon"};        // String

Tabell.innsettingssortering(a, (x,y) -> x.compareTo(y));        // lamda-uttrykk
Tabell.innsettingssortering(s, (x,y) -> x.compareTo(y));        // lamda-uttrykk

System.out.println(Arrays.toString(a));  // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
System.out.println(Arrays.toString(s));  // [Ann, Jon, Kari, Per, Sohil, Thanh]
```

Programkode* 1.4.7 *a)

I *Programkode* 1.4.7 *a)* brukes sorteringsmetoden fra *Programkode* 1.4.6 *b)*. Begge gangene brukes et og samme lambda-utrykk, dvs. `(x,y) -> x.compareTo(y)`. Variabelnavnene i et lambda-uttrykk kan vi velge helt fritt. Her har vi valgt de nøytrale navnene x og y. Det at x og y er av typen `Integer` i det første tilfellet og av typen `String` i det andre tilfellet, blir avledet av typen til tabellene, dvs. av a og s.

Konklusjon: Vi trenger kun sorteringsmetoden i *Programkode* 1.4.6 *b)*. Vi kan bruke det faste lambda-uttrykket `(x,y) -> x.compareTo(y)` for datatyper med «naturlig ordning». For andre ordninger eller andre datatyper, lager vi selv et lambda-uttrykk (dvs. en komparator).

Lambda-uttrykket `(x,y) -> x.compareTo(y)` er lett å huske, men vi kunne gjøre det enda enklere for oss. Vi kan lage en metode som gir oss lambda-uttrykket. Metoden hører naturlig hjemme i grensesnittet `Komparator`:

```
@FunctionalInterface
public interface Komparator<T>      // et funksjonsgrensesnitt
{
  int compare(T o1, T o2);          // en abstrakt metode

  public static <T extends Comparable<? super T>> Komparator<T> naturligOrden()
  {
    return (x, y) -> x.compareTo(y);
  }
}
```

Programkode* 1.4.7 *b)

Metoden *naturligOrden()* er en såkalt konstruksjonsmetode (eng: factory method). Den returnerer komparatoren gitt ved lambda-uttrykket `(x,y) -> x.compareTo(y)`. Dette går bra fordi datatyper som oppfyller typebegrensningen `<T extends Comparable<? super T>>` har metoden `compareTo`. Vi kan nå bruke dette i *Programkode* 1.4.7 *a)*:

```
Tabell.innsettingssortering(a, Komparator.naturligOrden());
Tabell.innsettingssortering(s, Komparator.naturligOrden());
```

Programkode 1.4.7 c)

Hvis vi ønsker å sortere i motsatt rekkefølge, kan vi selvfølgelig først sortere på vanlig måte og så snu tabellen. Men dette kan vi også få til ved å bytte om rekkefølgen i kallet på *compareTo*-metoden. Vi kan lage en konstruksjonsmetode for det også:

```
@FunctionalInterface
public interface Komparator<T>      // et funksjonsgrensesnitt
{
  int compare(T o1, T o2);          // en abstrakt metode

  public static <T extends Comparable<? super T>> Komparator<T> naturligOrden()
  {
    return (x, y) -> x.compareTo(y);
  }

  public static <T extends Comparable<? super T>> Komparator<T> omvendtOrden()
  {
    return (x, y) -> y.compareTo(x);
  }
}
```

Programkode 1.4.7 d)

Legg merke til at i naturligOrden() brukes x.*compareTo*(y), mens det i *omvendtOrden*() brukes y.*compareTo*(x). Flg. eksempel viser hvordan dette virker:

```
Integer[] a = {6,2,10,9,5,1,8,4,3,7};
String[] s = {"Sohil","Per","Thanh","Ann","Kari","Jon"};

Tabell.innsettingssortering(a, Komparator.omvendtOrden());
Tabell.innsettingssortering(s, Komparator.omvendtOrden());

System.out.println(Arrays.toString(a));  // [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
System.out.println(Arrays.toString(s));  // [Thanh, Sohil, Per, Kari, Jon, Ann]
```

Programkode 1.4.7 e)

Både *Programkode 1.4.6 e)* og *Programkode 1.4.6 g)* inneholder lambda-uttrykk. Hvis vi bruker x og y som argumenter begge steder, får lambda-uttrykkene flg. utseende:

```
(x,y) -> x.fornavn().compareTo(y.fornavn())
(x,y) -> x.studium().compareTo(y.studium())
```

I begge tilfellene har x og y metoder som returnerer en verdi som kan sammenlignes, dvs. har metoden *compareTo*(). Dette generaliserer vi ved hjelp av flg. funksjonsgrensesnitt:

```
@FunctionalInterface
public interface Funksjon<T,R>    // T for argumenttype, R for returtype
{
  R anvend(T t);
}
```

Programkode 1.4.7 f)

Funksjonsgrensesnittet Funksjon har metoden *anvend*(). Den har T som argumenttype og R som returverditype. Et lamda-uttrykk må ha et argument av typen *T* (f.eks. en Student) og en returverdi av en sammenlignbar type *R* (f.eks. String som er typen til metoden studium() i Student). Flg. konstruksjonsmetode (som legges i grensesnittet Komparator) gjør dette:

```
public static <T, R extends Comparable<? super R>>
Komparator<T> orden(Funksjon<? super T, ? extends R> velger)
{
  return (x, y) -> velger.anvend(x).compareTo(velger.anvend(y));
}
```

Programkode* 1.4.7 *g)

I lamda-uttrykket er *x* og *y* av typen *T*, mens metoden anvend() returnerer typen *R*. Også en Funksjon kan lages ved hjelp av et lamda-uttrykk. Vi bruker metoden studium() i Student som eksempel. Da vil Student svare til T og String til R:

```
Funksjon<Student,String> velger = x -> x.studium();
```

Her ser vi ikke metoden *anvend*() i Funksjon, men kun metodens «innmat». Argumentet x av typen Student går inn i metoden. Ut av metoden (som returverdi) går en String, dvs. tegnstrengen som instansmetoden studium() returnerer.

Dette kan gjøres enda enklere. Hvis klassen Student har kun én metode med navn studium, kan den hentes ved hjelp av flg. kode:

```
Funksjon<Student,String> velger = Student::studium;  // metodenavn etter ::
```

Nå kan vi bruke *Programkode* 1.4.7 *g)* til å lage en komparator:

```
Komparator<Student> c = Komparator.orden(velger);
```

Det er egentlig ikke nødvendig å lage funksjonen *velger* og komparatoren *c* eksplisitt. Vi kan forkorte det ned til flg. elegante kode (gitt at s er en tabell av studenter):

```
Tabell.innsettingssortering(s, Komparator.orden(Student::studium));
```

Programkode* 1.4.7 *h)

I *Programkode* 1.4.6 *i)* ble tegnstrenger (String) ordnet lengdevis - korte foran lange. Koden kan forenkles hvis vi bruker en *orden*:

```
String[] s = {"Lars","Anders","Bodil","Kari","Per","Berit"};
Tabell.innsettingssortering(s, Komparator.orden(String::length));

System.out.println(Arrays.toString(s));
// Utskrift: [Per, Kari, Lars, Berit, Bodil, Anders]
```

Programkode* 1.4.7 *i)

I *Programkode* 1.4.6 *j)* ble en samlig heltall (Integer) sortert leksikografisk, dvs. vi ser på sifrene som tegn. Vi prøver med en *orden* for det tilfellet:

```
Integer[] a = {13,25,11,3,2,21,10,1,33,100};  // en Integer-tabell
Tabell.innsettingssortering(a, Komparator.orden(Integer::toString));
```

Her vil det imidlertid komme en feilmelding. Integer::*toString* er ikke entydig. Klassen Integer har flere metoder med navn *toString*. Men det går bra hvis vi isteden bruker et

lambda-uttrykk. Der presiserer vi hvilken *toString*-metode som skal brukes. I flg. kodebit bruker vi samme Integer-tabell som over.

```
Tabell.innsettingssortering(a, Komparator.orden(x -> x.toString()));
System.out.println(Arrays.toString(a)); // [1, 10, 100, 11, 13, 2, 21, 25, 3, 33]
```

Programkode **1.4.7** *j)*

Istedenfor en sammenlignbar datatype *R* slik som i *Programkode* 1.4.7 *g)*, kan vi ha en type *R* med en komparator som virker for *R*. Flg. konstruksjonsmetode (som hører til grensesnittet Komparator) bruker den idéen:

```
public static <T, R> Komparator<T> orden
(Funksjon<? super T, ? extends R> velger, Komparator<? super R> c)
{
  return (x, y) -> c.compare(velger.anvend(x), velger.anvend(y));
}
```

Programkode **1.4.7** *k)*

Oppgaver til Avsnitt 1.4.7

1. La din versjon av grensesnittet Komparator inneholde alt det som er laget i *Avsnitt* 1.4.7, dvs. dette. Sørg for at det ligger under mappen (package) eksempelklasser. Flytt også grensesnittet Funksjon over til deg (legg det under eksempelklasser).
2. Sorter og skriv ut tabellen `Double[] d = {5.7,3.14,7.12,3.9,6.5,7.1,7.11};` Bruk en naturligOrden-komparator. Sorter så tabellen motsatt vei (minst til slutt).
3. Gitt `Boolean[] b = {false, true, true, false, false, true, false, true};` Bruk en naturligOrden-komparator til å sortere tabellen. Skriv ut resultatet.
4. Ta utgangspunkt i persontabellen *p* i *Programkode* 1.4.6 *c)*. Lag kode som sorterer kun med hensyn på etternavn. Bruk en *orden*-teknikk.
5. I *Programkode* 1.4.7 *i)* sorteres strenger mhp. lengde. Lag kode som sorterer motsatt vei, dvs. lange strenger kommer først. Lag en komparator eller bruk en *orden*-teknikk.
6. I *Programkode* 1.4.7 *j)* presiserer lambda-uttrykket hvilken toString-metode som skal brukes. Kan du få til samme effekt med en annen toString-metode fra klassen Integer? Sorter så tabellen leksikografisk mhp. binærkoden til heltallene. Skriv ut resultatet der hvert heltall skrives ut med sin binærkode. Sorter så tabellen mhp. lengden på tallenes binærkoder. Rekkefølgen på like lange binærkoder er likegyldig. Skriv ut resultatet binært. Hint: Bruk metoden i *Programkode* 1.4.7 *k)*.
7. Metoden *maks*() (se *Oppgave* 4 i *Avsnitt* 1.4.6) finner generelt posisjonen til den «største» verdien i en tabell. Bruk den til å finne største verdi i tabellen i *Oppgave* 2. Finn så den minste verdien (Hint: bruk komparatoren omvendtOrden).
8. I metoden *compare*() i en naturligOrden-komparator gjøres det et kall på datatypens compareTo-metode. Dermed vil trolig en sortering bli litt mindre effektiv ved å bruke komparatorversjonen enn den som benytter at datatypen er sammenlignbar. Lag kode som oppretter en tilfeldig *Integer*-tabell. Bruk f.eks. metoden i *Programkode* 1.4.3 *d)*. Sorter først tabellen ved hjelp av innsettingssorteringen i *Programkode* 1.4.2 *e)*. Sørg for at tabellen er så stor at det tar ca 10 sekunder. Bruk så *Programkode* 1.4.6 *b)* og naturligOrden(). Tar dette lenger tid?

1.4.8 Leksikografiske ordninger

I *Avsnitt* 1.4.6 så vi på hvordan vi ved hjelp av en komparator, kunne ordne personer etter fornavn og studenter etter studium. Det ble gjort ved hjelp av flg. lambda-uttrykk:

```
Komparator<Person>  c = (p1,p2) -> p1.fornavn().compareTo(p2.fornavn());
Komparator<Student> d = (s1,s2) -> s1.studium().compareTo(s2.studium());
```

Ved hjelp av en Funksjon og *Programkode* 1.4.7 *g)* kunne det kortes ned til:

```
Komparator<Person>  c = Komparator.orden(Person::fornavn);
Komparator<Student> d = Komparator.orden(Student::studium);
```

I begge disse tilfellene ordnes det kun etter én regel, mens de som er «like» med hensyn på denne regelen ikke får noen bestemt innbyrdes ordning. I *Programkode* 1.4.6 *h)* ble det laget en komparator som første ordnet studenter etter studium og som så ordnet de i på studium som personer (etternavn og fornavn). Det kalles en *leksikografisk* ordning. En slik ordning er en sammensetning av to eller flere ordninger eller regler. Hvis to verdier er «like» med hensyn på den første regelen, ordnes de etter den andre regelen. Hvis de også er «like» med hensyn på den andre regelen, ordnes de etter den tredje regelen. Osv.

Begrepet leksikografisk ordning er mest kjent i forbindelse med den vanlige ordningen av ord. F.eks. vil "Anders" komme foran "Andrine". De er like på de tre første bokstavene, men skiller seg på den fjerde bokstaven. Et spesialtilfelle er at et ord kan utgjøre første del av et annet ord. Da er det korteste ordet «minst». F.eks. kommer "Anders" foran "Andersen".

Det er ikke vanskelig å lage et lambda-utrykk som ordner leksikografisk. F.eks. vil flg. komparator ordne studenter først etter studium, så etter fornavn og til slutt etter etternavn:

```
Komparator<Student> c = (s1,s2) ->
{
  int k = s1.studium().compareTo(s2.studium());
  if (k != 0) return k;    // forskjellige studier
  k = s1.fornavn().compareTo(s2.fornavn());
  if (k != 0) return k;    // forskjellige fornavn
  return s1.etternavn().compareTo(s2.etternavn());
};
```

Programkode* 1.4.8 *a)

Et annet eksempel kunne være å ordne ord (tegnstrenger) først etter lengde og så alfabetisk hvis de er like lange:

```
Komparator<String> c = (s1,s2) ->
{
  int k = s1.length() - s2.length();
  return k != 0 ? k : s1.compareTo(s2);
};

String[] navn = {"Lars","Anders","Bodil","Kari","Per","Berit"};
Tabell.innsettingssortering(navn, c);

System.out.println(Arrays.toString(navn));
// Utskrift: [Per, Kari, Lars, Berit, Bodil, Anders]
```

Programkode* 1.4.8 *b)

Å «sette sammen» ordninger til en som er leksikografisk, er så vanlig at det kan lønne seg å ha en teknikk for det. Men dette har kun mening for *preordninger*, dvs. at metoden som sammenligner *x* og *y* returnerer 0 uten at *x* og *y* nødvendigvis er like. *Programkode* 1.4.8 *b)* inneholder en komparator som ordner tegnstrenger først etter lengde og så alfabetisk. To tegnstrenger kan være like lange uten at de er like. De to delene kan isteden settes opp med hver sin komparator:

```
Komparator<String> c1 = Komparator.orden(String::length);  // ordner etter lengde
Komparator<String> c2 = Komparator.naturligOrden();        // ordner alfabetisk
```

Vårt ønske kunne nå være å lage en sammensetning av de to og at det kunne gjøres slik:

```
Komparator<String> c  = c1.deretter(c2);      // sammensetingen av c1 og c2
```

Dette skal fungere slik at det først ordnes ved hjelp av `c1` og deretter ved hjelp av `c2` hvis `c1` gir 0. Det betyr at *deretter* må være en instansmetode til `c1`. Det får vi til ved å legge flg. metode inn i grensesnittet `Komparator`:

```
default Komparator<T> deretter(Komparator<? super T> c)
{
  return (x, y) ->
  {
    int k = compare(x, y);
    return k != 0 ? k : c.compare(x, y);
  };
}
```

Programkode* 1.4.8 *c)

Et grensesnitt kan ha *abstrakte* metoder, *statiske* metoder og *default* metoder. En default metode blir en instansmetode til en klasse (også en anonym klasse) som implementerer grensesnittet. Et lambda-uttrykk gir oss en instans av en anonym klasse. Hvis du har lagt *Programkode* 1.4.8 *c)* inn i grensesnittet `Komparator`, vil flg. kodebit virke:

```
Komparator<String> c1 = Komparator.orden(String::length);  // ordner etter lengde
Komparator<String> c2 = Komparator.naturligOrden();        // ordner alfabetisk

Tabell.innsettingssortering(navn, c1.deretter(c2));

System.out.println(Arrays.toString(navn));
// Utskrift: [Per, Kari, Lars, Berit, Bodil, Anders]
```

Programkode* 1.4.8 *d)

Dette kan kortes ned. De tre første kodelinjene i *Programkode* 1.4.8 *d)* kan erstattes med:

```
Tabell.innsettingssortering(navn,
  Komparator.orden(String::length).deretter(Komparator.naturligOrden()));
```

En komparator som ordner studenter først etter studium, så etter fornavn og til slutt etter etternavn, kan settes opp slik (sammenlign med *Programkode* 1.4.8 *a*):

```
Komparator<Student> c = Komparator.orden(Student::studium).
                        deretter(Komparator.orden(Student::fornavn)).
                        deretter(Komparator.orden(Student::etternavn));
```

Programkode* 1.4.8 *e)

Vi kan lage en komparator av typen i *Programkode* 1.4.8 *e)* på en enklere måte ved å bruke samme idé som i *Programkode* 1.4.7 *g)*, dvs. vi lar en funksjon velge en sammenlignbar verdi *R* for datatypen *T*. Det gjøres i flg. instansmetode (som hører til grensesnittet `Komparator`):

```
default <R extends Comparable<? super R>>  // tilhører grensesnittet Komparator
Komparator<T> deretter(Funksjon<? super T, ? extends R> velger)
{
  return (x, y) ->
  {
    int k = compare(x, y);
    return k != 0 ? k : velger.anvend(x).compareTo(velger.anvend(y));
  };
}
```

Programkode 1.4.8 *f)*

Nå kan komparatoren i *Programkode* 1.4.8 *e)* lages slik:

```
Komparator<Student> c = Komparator.orden(Student::studium).
                        deretter(Student::fornavn).
                        deretter(Student::etternavn);
```

Programkode 1.4.8 *g)*

Et morsomt eksempel på bruk av teknikken med leksikografisk ordning er å sortere navn som har en blanding av små og store bokstaver. Gitt navnene OLE, Per, Kari, PER, Ole, kari, per, KARI og ole. Internasjonal standard for sortering (se *Avsnitt* 1.4.10) sier at liten og stor bokstav er samme bokstav, dvs. *a* og *A* er det samme. Det betyr at *a/A* kommer foran *b/B*, osv. Men innbyrdes kommer liten bokstav foran stor, dvs. *a* kommer foran *A*, osv. Dermed blir dette rett sortering for de seks navnene:

kari, Kari, KARI, ole, Ole, OLE, per, Per, PER

Men hvis de sorteres på vanlig måte som tegnstrenger, får vi imidlertid flg. resultat:

```
String[] s = {"OLE","Per","Kari","PER","Ole","kari","per","KARI","ole"};

Tabell.innsettingssortering(s, Komparator.naturligOrden());
System.out.println(Arrays.toString(s));

// Utskrift: [KARI, Kari, OLA, Ola, PER, Per, kari, ola, per]
```

Programkode 1.4.8 *h)*

Klassen `String` har instansmetoden `compareToIgnoreCase()`. Den kan vi bruke til å la stor og liten bokstav være det samme. Da får vi alle forekomstene av Kari først, så alle av Ole, osv. Men rekkefølgen mellom de ulike variantene av samme navn blir tilfeldig:

```
Tabell.innsettingssortering(s, (x,y) -> x.compareToIgnoreCase(y));
System.out.println(Arrays.toString(s));

// Utskrift: [Kari, kari, KARI, OLE, Ole, ole, Per, PER, per]
```

Programkode 1.4.8 *i)*

Men, hvis vi starter slik som i *Programkode* 1.4.8 *i)* og *deretter* bruker omvendt naturlig orden, blir det riktig:

```
Komparator<String> c = (x,y) -> x.compareToIgnoreCase(y);
Tabell.innsettingssortering(s, c.deretter(Komparator.omvendtOrden()));

System.out.println(Arrays.toString(s));
// [kari, Kari, KARI, ole, Ole, OLE, per, Per, PER]
```

Programkode 1.4.8 j)

Java har en ferdig teknikk for resultatet i *Programkode* 1.4.8 *j)*. Der inngår en `Collator` (se *Avsnitt* 1.4.10). I følgende eksempel brukes sorteringsmetoden i klassen `Arrays` siden vår innsettingssortering foreløpig ikke kan brukes for en `Comparator` og dermed heller ikke for en `Collator`. Vi må ha `import java.text.Collator;` øverst for at flg. setning skal virke:

```
Arrays.sort(s, Collator.getInstance(Locale.US));  // finnes ingen norsk versjon
System.out.println(Arrays.toString(s));
// Utskrift: [kari, Kari, KARI, ole, Ole, OLE, per, Per, PER]
```

Programkode 1.4.8 k)

Vi kan også lage en instansmetode `deretter()` med samme idé som *Programkode* 1.4.7 *k)*. Dvs. at en funksjon `velger()` henter ut en verditype *R* fra *T* og der *R* sammenlignes ved hjelp av en komparator. Dette lages slik:

```
default <R> Komparator<T>
deretter(Funksjon<? super T, ? extends R> velger, Komparator<? super R> c)
{
  return (x, y) ->
  {
    int k = compare(x, y);
    return k != 0 ? k : c.compare(velger.anvend(x), velger.anvend(y));
  };
}
```

Programkode 1.4.8 l)

En komparator som «snur» en ordning, kan også lages som en default metode:

```
default Komparator<T> omvendt()
{
  return (x, y) -> compare(y, x);  // bytter x og y
}
```

Programkode 1.4.8 m)

Oppgaver til Avsnitt 1.4.8

1. Utvid din `Komparator` med det som er laget i *Avsnitt* 1.4.8, dvs. at den blir lik dette.
2. Blir `Komparator.orden(x -> x)` og `Komparator.naturligOrden()` det samme? Forklar!
3. Komparatoren i *Programkode* 1.4.6 *k)* ordner heltall (`Integer`) slik at oddetall kommer foran partall. Men innbyrdes har de ingen bestemt ordning. Bruk metoden `deretter()` slik at både oddetallene og partallene innbyrdes blir sortert stigende.
4. I *Programkode* 1.4.7 *i)* ble det laget en komparator for datatypen `String` som ordner etter lengde. Men strenger med samme lengde får ingen bestemt orden. Bruk metoden `deretter()` slik at strenger med samme lengde ordnes alfabetisk. Bruk dette til å sortere `s = {"21","18","8","13","20","6","16","25","3","10"};`
5. Som i *Oppgave* 4, men det skal ordnes slik at lange strenger kommer foran korte og de som er like lange skal ordnes alfabetisk synkende.

1.4.9 Ordninger - Comparable versus Comparator

Vi har laget grensesnittet Komparator. Som nevnt i innledningen til *Avsnitt 1.4.6*, har Java allerede grensesnittet Comparator. Der var det (i Java 1.7 og tidligere) kun én metode (den abstrakte metoden *compare*). I Java 1.8 er det fortsatt kun én abstrakt metode, men i tillegg har det kommet ni statiske og syv default metoder. I disse metodene inngår det som kalles *funksjonell programmering*. Det er en ny teknikk i Java og kanskje ikke helt enkelt å sette seg inn i. Derfor valgte vi å lage vår egen versjon. Det gav oss muligheten til å lære hvordan de ulike metodene er laget og hva de kan brukes til. Men nå som vi vet hvordan dette fungerer, skal vi isteden bruke Comparator. Tabellen under viser navneforskjellene:

Type	Navn i kompendiet	Navn i java.util
Grensesnitt	Komparator	Comparator
Grensesnitt	Funksjon	Function
Abstrakt metode	***compare***	***compare***
Statisk metode	***naturligOrden***	***naturalOrder***
Statisk metode	***omvendtOrden***	***reverseOrder***
Statisk metode	***orden***	***comparing***
Metodeargument	velger	keyExtractor
Default metode	***deretter***	***thenComparing***
Default metode	***omvendt***	***reversed***

Tabell 1.4.9 a) - Navn i kompendiet og i Java

Inntil nå kunne vi, hvis vi ønsket oss komparatorer for å ordne tegnstrenger på vanlig måte og med hensyn på lengde, gjøre det slik:

```
Komparator<String> c1 = Komparator.naturligOrden();             // vanlig orden
Komparator<String> c2 = Komparator.orden(String::length);       // etter lengde
```

Men fra nå av skal vi gjøre det slik:

```
Comparator<String> c1 = Comparator.naturalOrder();              // vanlig orden
Comparator<String> c2 = Comparator.comparing(String::length);   // etter lengde
```

Programkode 1.4.9 a)

I metoden *innsettingssortering* må bokstaven *K* i Komparator endres til *C* slik at det blir Comparator. Alt annet i metoden skal være som før. Signaturen blir da:

```
public static <T> void innsettingssortering(T[] a, Comparator<? super T> c)
```

Programkode 1.4.9 b)

Har du gjort denne endringen, vil flg. programbit virke. Prøv det!

```
String[] s = {"Sohil","Per","Thanh","Ann","Kari","Jon"};        // String-tabell
Comparator<String> c =  Comparator.comparing(String::length);  // etter lengde
Tabell.innsettingssortering(s, c.thenComparing(x -> x));        // vanlig orden
System.out.println(Arrays.toString(s));                         // skriver ut
```

Programkode 1.4.9 c)

Syntaksen for metoden *comparing*() som brukes i *Programkode* 1.4.9 *c)* over, er identisk med den for metoden *orden*() i *Programkode 1.4.7 g)*. Funksjonen som inngår som argument, skal ta imot en referansetype *T* og returnere en «sammenlignbar» referansetype *R*. Men i *Programkode* 1.4.9 *c)* inngår funksjonen *length*() fra klassen String som argument. Den returnerer en *int* og ikke en referansetype. Dette går likevel bra siden en *int* implisitt blir konvertert til *Integer* (såkalt autoboksing). Men det koster litt. Comparator har derfor fått en del metoder som unngår slike konverteringer. Et eksempel er comparingInt:

```
Comparator<String> c =  Comparator.comparingInt(String::length);
```

Det er tilsvarende metoder for *long* og *double*. Se Comparator.

Comparable: I overskriften på dette avsnittet står det *Comparable versus Comparator*. Gitt at vi skal lage en klasse der instanser vil kunne bli sammenlignet og ordnet. Skal vi da la den implementere Comparable (bli sammenlignbar) eller skal vi lage en Comparator som tar seg av ordningen? Her er det ingen fasit, men hvis det kan sies at instansene har en *naturlig* ordning, så er det vanlig å la klassen implementere Comparable. Tall har en naturlig ordning – de ordnes etter størrelse. Bokstaver og ord har også en naturlig ordning – de ordnes alfabetisk. Klokkeslett og datoer har naturlige ordninger – de ordnes i tidsrekkefølge. Personer kan sies å ha naturlig ordning – de ordnes normalt leksikografisk (etternavn og så fornavn). Det er ingen regel som forteller om en ordning er naturlig eller ikke. Det er ofte basert på skjønn. Men det er noen krav som må være oppfylt. Se nedenfor om *compareTo()*.

De forskjellige bibliotekene i Java 1.8 har hele 152 klasser som implementerer Comparable. Der inngår selvfølgelig klasser som Boolean, Byte, Short, Integer, Long, BigInteger, Float, Double, BigDecimal, Character, String, LocalTime, LocalDate og Calendar.

Comparator: Hvis instanser av en klasse som allerede er sammenlignbar (dvs. implementerer Comparable eller er en subtype til en slik en) skal ordnes på en spesiell måte, dvs. på en annen måte enn den «naturlige», må det lages en komparator for denne spesielle ordningen. Dette så vi eksempler på i *Avsnitt 1.4.6* der det ble laget komparatorer for å ordne personer etter fornavn, studenter etter klasse og tegnstrenger etter lengde.

Spesielt gjelder at hvis en klasse som ikke er sammenlignbar, skal ordnes på en en eller annen måte, så må vi lage en komparator for den ordningen. I følgende eksempel ser vi på punkter i et *x*, *y* - koordinatsystem. De har ingen naturlig ordning. Men det kan være måter å ordne dem på som er av interesse i spesielle situasjoner. F.eks. slik: Av to punkter er det «minst» som har minst *x* - koordinat. Hvis to punkter har like *x* - koordinater, er det «minst» som har minst *y* - koordinat, dvs. en leksikografisk ordning mhp. *x* - og *y* - koordinatene.

Vi bruker den ferdige klassen Point fra java.awt til representere punkter med kun heltallige koordinater. Vi lager først en komparator med direkte kode:

```
Comparator<Point> c = (p1, p2) ->
{
  int d = p1.x - p2.x;     // forskjellen mellom x-koordinatene
  if (d != 0) return d;
  return p1.y - p2.y;      // forskjellen mellom y-koordinatene
};
```

Programkode 1.4.9 d)

I *Programkode* 1.4.9 *d)* ser at hvis *x* - koordinatene er ulike, returneres differensen. Hvis de er like, returnereres differensene mellom *y* - koordinatene. Det betyr spesielt at vi får 0 som returverdi hvis og bare hvis de to punktene *p*1 og *p*2.

Men ved hjelp av teknikken fra avsnittene 1.4.7 og 1.4.8> (se også *Tabell 1.4.9 a*) kan vi lage en tilsvarende komparator som den i *Programkode* 1.4.9 *d)* på flg. enkle og elegante måte:

```
Comparator<Point> c = Comparator.comparing(Point::getX).thenComparing(Point::getY);
```

Programkode 1.4.9 e)

I klassen `Point` er koordinatene heltallige (*int*) og offentlige (public). Metodene *getX* og *getY* returnerer dem imidlertid som desimaltall (*double*). Her hadde det imidlertid mer effektivt å hente dem direkte (obs: vi må oppgi at typen til *p* er *Point* ellers blir det syntaksfeil):

```
Comparator.comparing((Point p) -> p.x).thenComparing(p -> p.y);
```

eller mer effektivt slik:

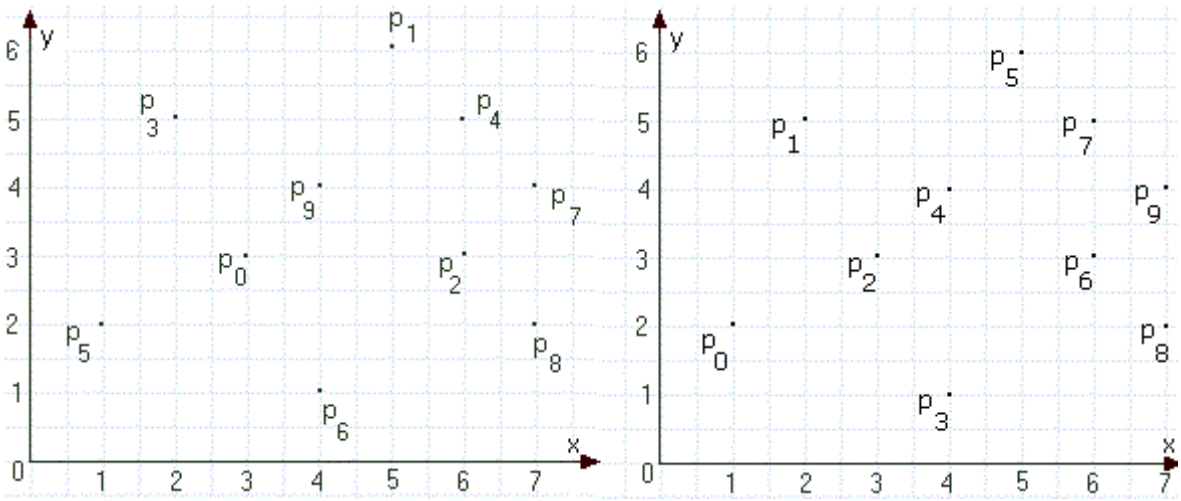
```
Comparator.comparingInt((Point p) -> p.x).thenComparingInt(p -> p.y);
```

Flg. eksempel viser hvordan dette kan brukes til å sortere punkter:

```
int[] x = {3,5,6,2,6,1,4,7,7,4};          // x-koordinater
int[] y = {3,6,3,5,5,2,1,4,2,4};          // y-koordinater

Point[] punkt = new Point[x.length];      // en punkttabell
for (int i = 0; i < punkt.length; i++) punkt[i] = new Point(x[i],y[i]);

for (Point p : punkt) System.out.print("(" + p.x + "," + p.y + ") ");
System.out.println();                     // linjeskift

Tabell.innsettingssortering(punkt,
  Comparator.comparing(Point::getX).thenComparing(Point::getY));

for (Point p : punkt) System.out.print("(" + p.x + "," + p.y + ") ");

// Utskriften blir:
// (3,3) (5,6) (6,3) (2,5) (6,5) (1,2) (4,1) (7,4) (7,2) (4,4)
// (1,2) (2,5) (3,3) (4,1) (4,4) (5,6) (6,3) (6,5) (7,2) (7,4)
```

Programkode 1.4.9 f)

I flg. figur er punktene nummerert før og etter sortering:



Figur 1.4.9 a): Punktenes nummerering - før sortering til venstre og etter sortering til høyre

Dato: Datoer har en naturlig ordning. Det er den leksikografiske etter tid, dvs. først etter år, så etter måned og til slutt etter dag. Flg. enkle datoklasse implementerer derfor *Comparable*:

```
public class Dato implements Comparable<Dato>
{
  private final int dag, mnd, år;              // instansvariabler

  public Dato(int dag, int mnd, int år)        // konstruktør
  {
    this.dag = dag; this.mnd = mnd; this.år = år;
  }

  public int compareTo(Dato d)                 // compareTo
  {
    if (år < d.år) return -1;
    else if (år > d.år) return 1;
    else if (mnd < d.mnd) return -1;
    else if (mnd > d.mnd) return 1;
    else return dag - d.dag;
  }

  public boolean equals(Object o)              // equals
  {
    if (o == this) return true;
    if (!(o instanceof Dato)) return false;
    return compareTo((Dato)o) == 0;
  }

  public String toString()                     // toString
  {
    return "" + dag + '/' + mnd + '-' + år;
  }

  public int hashCode()
  {
    return Objects.hash(dag,mnd,år);
  }
} // class Dato
```

Programkode* 1.4.9 *g)

Flg. eksempel viser hvordan datoklassen kan brukes:

```
Dato[] d = new Dato[5];           // en datotabell
d[0] = new Dato(24,12,2014);      // 24/12-2014
d[1] = new Dato(24,12,2012);      // 24/12-2012
d[2] = new Dato(9,12,2013);       // 9/12-2013
d[3] = new Dato(25,12,2012);      // 25/12-2012
d[4] = new Dato(10,12,2013);      // 10/12-2013

Tabell.innsettingssortering(d);
for (Dato x : d) System.out.print(x + " ");

// Utskrift: 24/12-2012 25/12-2012 9/12-2013 10/12-2013 24/12-2014
```

Programkode* 1.4.9 *h)

Oppsummering:

• ***compareTo***: Hvis instanser av en klasse *X* har en «naturlig» ordning, så er det som nevnt over, vanlig å la *X* implementere `Comparable<X>`. Da må det være flg. sammenheng mellom *compareTo* og ordningen (*x* og *y* er to vilkårlige instanser av klassen *X*):

1. *x* er «mindre enn» *y* hvis og bare hvis `x.compareTo(y) < 0`
2. *x* er «mindre enn eller lik» *y* hvis og bare hvis `x.compareTo(y) <= 0`
3. *x* er lik *y* hvis og bare hvis `x.compareTo(y) = 0`
4. `x.compareTo(y) < 0` hvis og bare hvis `y.compareTo(x) > 0`

Det følger av punktene at `x.compareTo(y) <= 0` hvis og bare hvis `y.compareTo(x) >= 0`. Det er vanligvis fordelaktig, men ikke et krav, at *compareTo* kodes slik at for alle *x* og *y* gjelder at `x.compareTo(y) = -y.compareTo(x)`. I punktene inngår kun «mindre enn» og «mindre enn eller lik», men som vanlig defineres *x* «større enn» *y* som at *y* er «mindre enn» *x*. Tilsvarende er det for «større enn eller lik».

Vi må også passe på metoden *equals*. Den må synkroniseres med *compareTo* siden begge kan brukes til å teste om to verdier er like. Kravet er at `x.equals(y) = true` hvis og bare hvis `x.compareTo(y) = 0`. Dette bør igjen synkroniseres med *hashCode*. Hvis *equals* sier at *x* og *y* er like, skal *hashCode(x) = hashCode(y)*. Se også *Avsnitt* 1.4.12.

• ***compare***: `Comparator` har metoden *compare*, men når vi lager en komparator ved hjelp av et lambda-uttrykk, ser vi ikke den. Lambda-uttrykket er på formen `(x,y) -> f(x,y)` der *f* er en funksjon som returnerer et heltall. Komparatorordningen behøver ikke være en *total ordning*. Det holder at den er en *total preordning*. Flg. krav til sammenhengen mellom *f* og ordningen må være oppfylt:

1. *x* er «mindre enn» *y* hvis og bare hvis `f(x,y) < 0`
2. *x* er «mindre enn eller lik» *y* hvis og bare hvis `f(x,y) <= 0`
3. hvis *x* er lik *y*, så er `f(x,y) = 0`
4. `f(x,y) < 0` hvis og bare hvis `f(y,x) > 0`

Det er vanligvis fordelaktig, men ikke et krav, at for alle *x* og *y* er $f(x,y) = -f(y,x)$. Det er spesielt punkt 3 som skiller dette fra en *total ordning*. Vi kan ha at `f(x,y) = 0` uten at *x* er lik *y*. Ta lamda-uttrykket i *Programkode* 1.4.6 *i)* som eksempel. Der er *f* gitt ved følgende uttrykk: `f(x,y) = x.length() - y.length()` der *x* og *y* er av typen `String`. Vi ser at `f(x,y) = 0` så sant *x* og *y* er like lange. Men det behøver på ingen måte bety at *x* = *y*. Denne spesielle ordningen er en *total preordning*, men ikke en *total ordning*. I en slik ordning vil forskjellige elementer *x* og *y* med `f(x,y) = 0` ikke ha noen spesiell innbyrdes ordning.

Oppgaver til Avsnitt 1.4.9

1. Gjør om fra `Komparator` til `Comparator` i metoden i *Programkode* 1.4.6 *b)*. Det eneste som trengs er å bytte ut `K` med `C` i metodens signatur. Se også *Programkode* 1.4.9 *b)*. Sjekk så at *Programkode* 1.4.9 *c)* virker. Gjør det samme i `maks`-metoden hvis du har laget en versjon av den der `Komparator` inngår (se *Oppgave* 4 i *Avsnitt* 1.4.6).
2. Lag en komparator-versjoner (`Comparator`) av a) `utvalgssortering`, b) `binærsøk`, c) `kvikksortering` og d) `flettesortering`. Legg metodene i samleklassen `Tabell`.
3. a) Sjekk at *Programkode* 1.4.9 *f)* virker og gir den utskriften som står oppgitt.

   b) Bruk komparatoren fra *Programkode* 1.4.9 *d)* i *Programkode* 1.4.9 *f)*. Virker det?

   c) De to komparatoren rett under *Programkode* 1.4.9 *e)* bruker koordinatene direkte. Bruk dem i *Programkode* 1.4.9 *f)*. Virker det?

d) Komparatorene i punkt c) over har `(Point p) -> p.x` i første parentes. Ta vekk `Point` slik at det kun står `p -> p.x`. Hva skjer?

e) Komparatoren i *Programkode* 1.4.9 *f)* bruker metodene *getX* og *getY* som returnerer en *double*. Deretter blir *double* konvertert til *Double* (autoboksing). Her får vi mer effektiv kode hvis vi isteden bruker `comparingDouble` og `thenComparingDouble`. Prøv det!

f) Anta at vi har en samling punkter i 1. kvadrant (ingen negative koordinater). Lag en komparator som ordner punktene på flg. måte: Et punkt som ligger nærmere origo enn et annet punkt regnes som «mindre» enn det andre. Hvis de ligger like langt fra origo regnes det som har minst *y*-koordinat som «minst». Bruk den i *Programkode* 1.4.9 *f)*.

g) Anta at vi har en samling punkter i 1. kvadrant (ingen negative koordinater). Vinkelen til et punkt *p* forskjellig fra origo defineres som den vinkelen den rette linjen gjennom origo og *p* danner med *x*-aksen. Lag en komparator som ordner punktene på flg. måte: Det av to punkter som har minst vinkel regnes som «minst». Hvis de to punktene har samme vinkel, regnes det som minst som ligger nærmest origo. Spesielt gjelder at origo, dvs. punktet (0,0), er «mindre enn» alle andre punkter. Bruk den i *Programkode* 1.4.9 *f)*.

4. a) Flytt `Dato` til deg (package eksempelklasser). Sjekk at *Programkode* 1.4.9 *h)* virker.

   b) I *Programkode* 1.4.9 *h)* inngår den versjonen av *innsettingssortering* som er laget for sammenlignbare typer, se *Programkode* 1.4.2 *e)*. Bruk isteden komparatorversjonen (*Programkode* 1.4.6 *b*) og en `naturalOrder`-komparator.

   c) Sjekk at *compareTo()* i datoklassen i *Programkode* 1.4.9 *g)* oppfyller kravene 1 - 4. Er det slik at `x.compareTo(y) = -y.compareTo(x)`?

   d) Metoden *equals()* i datoklassen er kodet ved hjelp av *compareTo()*. Lag egen kode for *equals()*. Pass på at koden blir slik at `x.equals(y) = y.equals(x)` for alle datoer *x* og *y*. Pass også på at `x.equals(y) = true` hvis og bare hvis `x.compareTo(y) = 0`.

   e) *hashCode()* i `Dato` er laget ved hjelp av en metode fra klassen `Objects`. Sjekk hva den gjør. Kommentér vekk så *hashCode()*. Bruker du NetBeans vil du ved siden av *equals()* få en melding (et gult symbol) om at du bør kode *hashCode()*. Klikker du på det gule symbolet, vil du få hjelp. Gjør det! Omvendt, har laget *hashCode()*, men ikke *equals()*, vil du også få hjelp. Det finnes tilsvarende muligheter i Eclipse - bruk menyvalget `Source`.

   f) Legg inn en ekstra konstruktør i datoklassen. I den skal måneden kunne skrives vha. enumkonstanter fra `enum Måned` (se *Oppgave* 3 i *Avsnitt* 1.4.5). For eksempel skal 17. mai og julaften kunne opprettes ved: `Dato mai17 = new Dato(17,Måned.MAI,2016);` og `Dato julaften = new Dato(24,Måned.DES,2016);` Uttskrift skal bli slik: 17. mai 2016 og 24. desember 2016. Lag konstruktøren og de tilleggene og endringene som må til. Bruk `StringBuilder` når du omkoder *toString()*.

5. Lag klassen `Klokkeslett`. Den skal implementere `Comparable`. Det holder med timer og minutter. Klokkeslett skal ordnes slik at 00:00 kommer først og 23:59 sist. Konstruktøren skal ha en `String` som argument og være på formen *tt:mm*, f.eks. "12:00". Lag også metodene *equals()*, *hashCode()* og *toString()*. Bruk `Dato`-klassen som mønster. Lag en programbit der du oppretter en tabell av klokkeslett som så sorteres og skrives ut.

6. Lag en klasse `Tid` som har et klokkeslett og en dato som instansvariabler. Se *Oppgave* 3 og *Programkode* 1.4.9 *g)*. Klassen skal implementere `Comparable`. Tider skal først ordnes mhp. dato og så med hensyn på klokkeslett. Lag også metodene *equals*, *hashCode* og *toString*. Benytt flest mulig av metodene i klassene `Dato` og `Klokkeslett` i kodingen.

1.4.10 Behandling av Æ, Ø og Å

Når en på norsk skal ordne eller sortere navn, ord og generelt tegnstrenger, må en blant annet ta hensyn til at:

- De «skandinaviske» bokstavene Æ (æ), Ø (ø) og Å (å) ikke er plassert sammen med de «latinske» bokstavene i tegnsettene.
- Den «norske» bokstaven Å (å) står «feil» i forhold til Æ (æ) og Ø (ø) i tegnsettene, f.eks. i ISO-8859-1, UTF-8 og Unicode.
- Mange steder (først og fremst i navn), men ikke alltid, skal aa sorteres som å.
- I mange navn med utenlandsk opprinnelse forekommer bokstaver som ä, ö og ü.
- Nasjonal (og internasjonal standard) sier at liten bokstav skal komme foran stor.

Bokstavene fra A - Z (og a - z) kalles de latinske bokstavene og brukes i alle «vestlige» språk og i andre språk. Det er en lang historie som ligger bak bokstavenes oppkomst, rekkefølge, utseende og uttale. I den elektroniske databehandlingens «barndom» ble det nødvendig å kunne representere dem (og andre tegn enn bokstavene) på en standardisert form i binært format. Den første standarden kom i 1963 og ble kjent som ASCII (American Standard Code for Information Interchange). Den bestod av 128 tegn og hvert tegn hadde en binærkode på 7 biter. Men dette foregikk, som navnet sier, i USA og dermed var det ingen som tenkte på bokstavene Æ, Ø og Å og andre bokstaver som brukes rundt i verden. På engelsk brukes som kjent kun bokstavene fra A - Z (og a - z).

Tallverdiene for A - Z går fra 65 - 90 og for a - z fra 97 - 122. Mellom Z og a ligger de seks tegnene [\ ] ^ _ `. På norsk valgte man å la de tre første av dem bli erstattet med Æ, Ø og Å. Med andre ord tok Æ plassen til [, osv. Tilsvarende ble de tre tegnene { | } som kommer etter z, erstattet med æ, ø og å. Denne tilpasningen av ASCII til norsk fikk forøvrig navnet ISO-646-60. Dette fungerte på et vis, men skapte ofte problemer. I programmering brukes jo disse seks tegnene hyppig og dermed måtte en lære seg å lese (og skrive) programkode (på skjerm eller på papir) der f.eks. Æ står der man forventet tegnet [. Et eksempel er at

```
æaÆiÅ = 'Øn';å
```

da blir det samme som

```
{a[i] = '\n';}
```

I ASCII er hvert tegn representert med 7 biter. Men internt i datamaskinene ble det normalt operert med byter på 8 biter og slik er det fortsatt. Dermed var det mulig å lage en «utvidet» ASCII. De opprinnelige 128 tegnene kunne få en ekstra 0-bit forrest og de 128 mulige som har en 1-bit forrest, kunne brukes til tegn fra andre alfabeter enn det engelske. Disse utvidelsene fikk fellesnavnet ISO-8859. Problemet var imidlertid at det ikke var plass til alle aktuelle tegn. Dermed ble det en familie av utvidelser der f.eks. ISO-8859-1 kalles Latinsk alfabet nr. 1. Det dekket behovet til vest-europeiske språk og dermed de skandinaviske. Første versjon av disse standardene kom i 1987. Et viktig poeng er at de 32 posisjonene med tallverdier fra 128 - 159 ikke ble tatt i bruk. Husk at heller ikke de 32 første (0 - 31) hører til vanlige tegn. De kalles kontrolltegn og bare noen få av dem brukes i praksis i dag.

De tegnene som mangler i ASCII er i hovedsak ulike varianter av de latinske vokalene A, E, I, O og U. De fleste språk har flere vokallyder. Det løses enten med ekstra symboler på vokalene for å signalisere at de skal ha spesiell uttale eller med nye vokaler laget ved å kombinere to andre. Et eksempel på det siste er Æ som er A + E. En slik konstruksjon kalles en *ligatur*. Et annet eksempel er Œ (O + E) som kan forekomme i fransk.

Et ekstra symbol/merke over (eller under) en bokstav kalles et *diakritisk* tegn. Det kan være et aksenttegn (*gravis* som i À, *akutt* som i Á eller *cirkumfleks* som i Â) eller en *tilde* (som i Ã). De to bokstavene Ä og Å sies også å ha diakritiske tegn, men historisk sett er de mer å regne som ligaturer. Bokstaven Å skal opprinnelig ha vært skrevet som to A-er der den andre var liten og skrevet over den første. Dette skal så ha utviklet seg til en ring. Mens Ä skal ha vært skrevet som en A med en liten e over. Så skal e-en ha utviklet seg til to prikker.

På norsk regnes ikke À, Á, Â og Ã som egne bokstaver. Men det gjør Å og Ä. Den siste hører ikke til det norske alfabetet, men i det svenske alfabetet spiller den omtrent samme rolle som vår Æ. Oslo-områdets nabofylke (svensk: län) i Sverige heter Värmland og på norsk skal normalt utenlandske egenavn skrives som de er.

Da ASCII ble utvidet til ISO-8859-1, fikk alle «variantene» av A egne plasser (og dermed 8-biters koder) og ble satt ved siden av hverandre. Deretter kom Æ siden den ble sett på som en mellomting mellom A og E.

Tegn/bokstav:	À	Á	Â	Ã	Ä	Å	Æ	··	Ö	Œ	Ø
Tallverdi:	192	193	194	195	196	197	198		214	215	216

Denne rekkefølgen gir sorteringsproblemer på norsk. Vi ser at Å ligger feil i forhold til Æ. På norsk er rekkefølgen Æ, Ø, Å. Men dette gir også problemer på svensk. De har rekkefølgen Å, Ä, Ö der Ä og Ö svarer til Æ og Ø på norsk. Med andre ord er ligger Å og Ä feil for dem. Disse sorteringsproblemene (på norsk, dansk og svensk) har vi fortsatt siden A-variantene har samme rekkefølge (og tallverdier) i f.eks. Unicode og UTF-8. Se flg. Javakode:

```
char[] c = {'Æ', 'Ø', 'Å'};
Arrays.sort(c);
System.out.println(Arrays.toString(c));  // Utskrift: [Å, Æ, Ø]
```

Det er litt underlig at norsk (og dansk) har rekkefølgen Æ, Ø, Å, mens den er Å, Ä, Ö på svensk. Det har imidlertid historiske årsaker. På svensk ble bokstaven Å tatt i bruk for svært lenge siden. De tre bokstavene kommer fra AA, AE og OE og dermed er Å, Ä (Æ), Ö (Ø) en naturlig rekkefølge. Men på norsk (og dansk) er Å relativt sett en ny bokstav. Den ble innført offisielt i 1917 (på dansk først i 1948). Før den tiden ble AA brukt for den vokallyden som svarer til dagens Å. Siden den var en ny bokstav, var det naturlig å legge den sist i alfabetet og dermed bak Æ og Ø som var der fra før.

Fortsatt brukes AA (Aa og aa) i navn - først og fremst i personnavn, men også i stedsnavn. I telefonkatalogen finner vi f.eks. både *Åserud* og *Aaserud*. Dette gir oss et problem ved sortering. I telefonkatalogen står det: «Navn som er skrevet med *aa* og uttales med *å*, står sammen med navn som er skrevet med *å*. Når *aa* uttales som *lang a*, er navnet å finne på alfabetisk plass foran ord skrevet med *enkel a*».

En kollator. Java har innført begrepet *kollator* (eng: collator). Det er en spesiell komparator for sammenligning og ordning etter egendefinerte regler. Substantivet *collator* kommer fra verbet *collate* som bl.a. kan bety å sammenligne tekst. Ordet *kollasjon* brukes på norsk og kan bety en «sammenligning av en avskrift med originalen» eller en «gjennomgang av en bok for å undersøke om den er komplett».

Java inneholder kollatorer for flere språk og da selvfølgelig for engelsk. Flg. eksempel viser hvordan en sortering av de første og siste bokstavene i alfabetet vårt vil slå ut avhengig av hva slags komparator som blir brukt (obs: må ha `import java.text.*;` øverst):

```
String[] s = {"A","B","C","a","b","c","Æ","Ø","Å","æ","ø","å"};

Arrays.sort(s,String.CASE_INSENSITIVE_ORDER);     // en komparator
System.out.println(Arrays.toString(s));
// [A, a, B, b, C, c, Å, å, Æ, æ, Ø, ø]          // utskrift

Arrays.sort(s);                                   // vanlig sortering
System.out.println(Arrays.toString(s));
// [A, B, C, a, b, c, Å, Æ, Ø, å, æ, ø]          // utskrift

Arrays.sort(s, Collator.getInstance(Locale.US)); // en US-kollator
System.out.println(Arrays.toString(s));
// [a, A, å, Å, æ, Æ, b, B, c, C, Ø, ø]          // utskrift
```

Programkode* 1.4.10 *a)

Java bruker Unicode og bokstavene har samme plassering der som i ISO-8859-1. I koden over brukes først komparatoren `CASE_INSENSITIVE_ORDER`. Den ser på liten og stor bokstav (f.eks. A og a) som det samme uten noen bestemt innbyrdes rekkefølge. I utskriften kommer stor bokstav først, men det kommer av at sorteringsmetoden er stabil. Se *Oppgave* 2.

I vanlig sortering kommer bokstavene i samme rekkefølge som de ligger i ISO-8859-1. Det betyr selvfølgelig at Å (og å) og de andre kommer i feil rekkefølge for oss.

I siste del av *Programkode* 1.4.10 *a)* inngår en engelsk (US) kollator. Internasjonal standard for sortering sier at A og a er samme bokstav, men innbyrdes skal a komme foran A. Denne kollatoren ser på Æ og Å som varianter av A og de kommer derfor mellom A og B i sorteringen. Videre er det en feil i kollatoren siden vi egentlig skal ha ø foran Ø.

Den abstrakte klassen `Collator` har en konkret subklasse med navn `RuleBasedCollator`. Setningen `Collator.getInstance(Locale.US)` returnerer en instans av den klassen. Den kan også brukes til å sortere etter en egendefinert regel. En regel må innholde informasjon om hvilken rekkefølge de ulike bokstavene skal ha. Vi kan f.eks. lage en regel som sørger for at bokstavene i *Programkode* 1.4.10 *a)* blir sortert slik de skal på norsk:

```
String[] s = {"A","B","C","a","b","c","Æ","Ø","Å","æ","ø","å"};

String regel = "< a < A < b < B < c < C < æ < Æ < ø < Ø < å < Å";
Arrays.sort(s, new RuleBasedCollator(regel));  // kan kaste ParseException

System.out.println(Arrays.toString(s));
// Utskrift: [a, A, b, B, c, C, æ, Æ, ø, Ø, å, Å]
```

Programkode* 1.4.10 *b)

I en «regel» brukes ulikhetstegn til å gi bokstavene rett rekkefølge. A og a ses på som samme bokstav i forhold til de andre bokstavene, men innbyrdes skal a komme foran A. Relasjonen a < A definerer egentlig a og A som ulike bokstaver. En mer korrekt måte er derfor å bruke relasjonen a , A. De andre bokstavene må settes opp tilsvarende. Det gir flg. regel:

```
String regel = "< a,A < b,B < c,C < æ,Æ < ø,Ø < å,Å";
```

Programkode* 1.4.10 *c)

Hvis regelen over brukes i *Programkode* 1.4.10 *b)* vil sorteringen gi samme resultat som før. Se *Oppgave* 4. Men det er likevel en ganske viktig forskjell mellom de to reglene. F.eks. vil

`c.compare("bBbB","BbBb")` gi -1 som resultat for begge reglene. Det betyr at "bBbB" er «mindre enn» "BbBb". Derimot blir det forskjell for `c.compare("bBbBb","BbBba")`. Da får vi fortsatt −1 for den første regelen, men 1 for den andre. Det kommer av at for den andre regelen er de første fire bokstavene like og avgjørelsen tas på den femte bokstaven. Siden b kommer etter a, blir resulatet 1. Se *Oppgave* 5.

Neste «problem» på norsk er aa kontra å. Nå kan aa forekomme inne i et ord uten at det skal tolkes som å. F.eks. kan en ansatt som arbeider med data, omtales som *dataansatt*. Vi ser imidlertid bort fra det problemet her. Neste spørsmål er om aa og å skal ses på som samme bokstav. Er f.eks. *Åserud* og *Aaserud* egentlig samme navn? I telefonkatalogen forekommer begge navnene. Men der er det i tillegg fornavn og de bestemmer rekkefølgen. Ta f.eks. Per Åserud, Kari Aaserud og Elin Åserud. I telefonkatalogen vil de komme i denne rekkefølgen: Elin Åserud, Kari Aaserud, Per Åserud siden rekkefølgen på fornavnene er Elin, Kari, Per. Vi har to muligheter for aa og å. Bruker vi relasjonen `aa = å` defineres de som like. Da vil `c.compare("aa","å")` gi 0. Det samme vil skje hvis vi sammenligner *Åserud* og *Aaserud*. Men vi kan også bruke relasjonen `aa ,å`. Da vil `c.compare("aa","å")` gi -1.

Vi velger her at aa og å skal ses på som samme bokstav (dvs. relasjonen `aa = å`) og det samme for Aa og Å, men at aa skal komme foran Aa (stor og liten bokstav). Dermed blir det slik i regelen: `å = aa, Aa = Å`. Flg. kodebit viser hvordan dette virker:

```
String[] s = {"Åserud,Per", "Aaserud,Kari", "Åserud,Elin"};

String regel = "< a,A < b,B < c,C < æ,Æ < ø,Ø < å = aa,Aa = Å";
Arrays.sort(s, new RuleBasedCollator(regel));

System.out.println(Arrays.toString(s));
// Utskrift: [Åserud,Elin, Aaserud,Kari, Åserud,Per]
```

Programkode* 1.4.10 *d)

Rekkefølgen for tegn i en regel bestemmes ved hjelp av relasjoner:

1. `<` er den primære ulikheten. Skiller tegn. F.eks. A < B
2. `;` er den sekundære ulikheten. Skiller mellom aksenter. F.eks. À ; Á
3. `,` er den tertiære ulikheten. Skiller mellom liten og stor bokstav. F.eks. a , A
4. `=` er identiteten. To tegn ses på som samme tegn. F.eks. Aa = Å

Den tertiære ulikheten (;) kan f.eks. brukes slik:

```
String regel = "< a;à;á;â;ã , A;À;Á;Â;Ã < b,B";
```

De «svenske» bokstavene Ä (ä) og Ö (ö) inngår ikke i det norske alfabetet, men en regel sier at f.eks. personnavn og stedsnavn som inneholder disse bokstavene, skal skrives på sin originale form. Derfor tar vi dem med i sorteringsregelen (a - y mangler) (se *Oppgave* 6):

```
String norsk = " . . < z,Z < ä = æ,Æ = Ä < ö = ø,Ø = Ö < å = aa,Aa = Å";
```

Programkode* 1.4.10 *e)

I Java brukes Unicode for tegn. I noen programmeringsmiljøer kan det være vanskelig å få skrevet inn «uvanlige» tegn i klartekst. Da kan man oppgi dem i Unicode-format. F.eks. '\u00E5' istedenfor 'å' og '\u00C5' istedenfor 'Å'. Bokstavens tallverdi må oppgis med med fire heksadesimale siffer. I Unicode kan også 'å' (og 'Å') skrives som «a + ring» der ring er tegnet '\u030A'. Dermed kunne siste delen av den «norske» regelen skrives slik:

```
String norsk = " . . < \u00E5 = a\u030A = aa,Aa = \u00C5 = A\u030A";
```

Relasjonene i en regel har ulik «styrke». Den «sterkeste» (primære) er ulikheten <. Styrken kan endres uten endringer i selve regeldefinisjonen (tekststrengen). Sammenligningene utføres av komparatorens compare-metode. Se flg. eksempel:

```
String regel =  "< a,A ; à,À < b,B < c,C < æ,Æ < ø,Ø < å = aa,Aa = Å";
Collator c = new RuleBasedCollator(regel);
c.setStrength(Collator.PRIMARY);

System.out.println(c.compare("a","A"));  // Utskrift: 0
```

Programkode* 1.4.10 *f)

Metoden compare ser nå på a og A som identiske tegn. Det å sette styrken til PRIMARY (lik 0) fører til at alle bokstavene mellom to ulikhetstegn blir sett på som identiske. Med andre ord vil compare gi 0 uansett hvilke to av bokstavene a, A, à, À som sammenlignes.

Vi kan sette styrken til SECONDARY (lik 1). Da vil fortsatt a og A være identiske, men de vil begge komme foran à og À. Det betyr at relasjonen (;) forsatt virker. Når en kollator opprettes blir styrken automatisk satt til TERTIARY (lik 2).

Det finnes flere andre teknikker som kan brukes for å bygge opp og endre på en regel. Dette er beskrevet i API-en til RuleBasedCollator. Der står det også et utkast til en norsk regel.

Oppgaver til Avsnitt 1.4.10

1. Kjør *Programkode* 1.4.10 *a)* og sjekk at utskriften blir slik det påstås.
2. På svensk er rekkefølgen Å, Ä og Ö Hva blir resultatet hvis *Programkode* 1.4.10 *a)* bruker `s = {"A","B","C","a","b","c","Å","Ä","Ö","å","ä","ö"}`.
3. Kjør *Programkode* 1.4.10 *b)* og sjekk at utskriften blir slik det påstås.
4. Bruk regelen `"< a,A < b,B < c,C < æ,Æ < ø,Ø < å,Å"` i *Programkode* 1.4.10 *b)*.
5. Gitt tabellen `{"A","a","aC","AaB","aAB"}`. Sorter ved hjelp av `regel1 = "< a < A < b < B"` og så med `regel2 = "< a,A < b,B"`. Hva som skjer?
6. Lag ferdig den norske regelen i *Programkode* 1.4.10 *e)*, dvs. ta med bokstavene (små og store) fra a - y. Ta også med Ü (ü). Den skal være identisk med Y (y). Ta også med de bokstavene med aksenttegn som normalt dukker opp i norsk. F.eks. trenger vi é siden den f.eks. er med i kafé. Bokstaven ô brukes i ordet fôr (i betydningen dyrefôr). Lag testprogram der du sorterer etter denne regelen. Lag så klassen `Kollator`. Der legges den norske regelen som en statisk tegnstreng. Klassen skal ha en statisk metode `norsk()` som returnerer en `Collator` ved hjelp av `RuleBasedCollator`.
7. Konsekvensen av å definere at aa = å (og Aa = Å) blir at ord som ellers er like (f.eks. Aaserud og Åserud), blir sortert i en tilfeldig rekkefølge. Sorter f.eks. tabellen `String[] s = {"Aa","å","aa","Å","aa","Å","å","Aa"};`
8. Lag kode som sorterer "datamaskin", "datasjef", "dataansatt" og "datafreak" etter norsk regel. Hva blir resultatet?
9. Sjekk hvordan tegn som ikke inngår i en regel, blir sortert, f.eks. sifrene fra 0 - 9. Sorter tabellen `{"A","6","B","5","C","4","Æ","3","Ø","2","Å","1"}`. Utvid så regelen *norsk* slik at sifrene 0 - 9 kommer foran a,A.

1.4.11 Generelt om ordninger

Begrepet *ordning* defineres vanligvis ved hjelp av begrepet *relasjon*. La A være en mengde og R en delmengde av produktmengden $A \times A$. Da definerer R en relasjon på A. La a og b være to elementer i A. Da sier vi at a er relatert til b hvis det ordnede paret (a, b) er element i mengden R, dvs. $(a, b) \in R$. Det at a er relatert til b kan også skrives som $a\,R\,b$.

Flg. begreper er viktige for relasjoner:

1. *Diagonalen D* i produktmengden $A \times A$ består av alle par (a, a) der $a \in A$.
2. *Den inverse relasjonen* Den inverse relasjonen til en relasjon R betegnes med R^{-1}. Den er definert ved at $(a, b) \in R^{-1}$ hvis og bare hvis $(b, a) \in R$.
3. *Refleksivitet* En relasjon R kalles *refleksiv* hvis alle elementer i A er relatert til seg selv, dvs. $(a, a) \in R$ for alle elementer $a \in A$. Alternativt: En relasjon R er refleksiv hvis diagonalen D er inneholdt i R, dvs. $D \subseteq R$.
4. *Antirefleksivitet* En relasjon R kalles *antirefleksiv* hvis ingen elementer i mengden A er relatert til seg selv, dvs. for alle $a \in A$ er $(a, a) \notin R$. Alternativt: En relasjon R er antirefleksiv hvis $R \cap D = \emptyset$.
5. *Symmetri* En relasjon R kalles *symmetrisk* hvis det for alle par av elementer $a, b \in A$ er slik at hvis $(a, b) \in R$, så er også $(b, a) \in R$. Alternativt: En relasjon R er symmetrisk hvis $R = R^{-1}$.
6. *Antisymmetri* En relasjon R kalles *antisymmetrisk* hvis det for alle forskjellige elementer a og b i A, dva. $a \neq b$, er slik at hvis $(a, b) \in R$, så er $(b, a) \notin R$. Alternativt: En relasjon R er antisymmetrisk hvis $R \cap R^{-1}$ er inneholdt i diagonalen D.
7. *Transitivitet* En relasjon R kalles *transitiv* hvis det for alle elementer a, b og c i A er slik at hvis $(a, b) \in R$ og $(b, c) \in R$, så er $(a, c) \in R$.
8. *Sammenlignbarhet* La R være en relasjon på A. To elementer a og b kalles *sammenlignbare* hvis $(a, b) \in R$ eller $(b, a) \in R$.
9. *Totalitet* En relasjon R kalles *total* hvis alle elementer a og b i A er sammenlignbare. Alternativt: En relasjon R er total hvis $R \cup R^{-1} = A \times A$.
10. *Ekvivalensrelasjon* En relasjon R kalles en *ekvivalensrelasjon* hvis den er refleksiv, transitiv og symmetrisk. I så fall sier vi at a og b er ekvivalente hvis a er relatert til b (da er også b relatert til a). Ekvivalensklassen til a består av de elementene som er ekvivalent med a. To ekvivalensklasser er enten like eller disjunkte.

For at en *relasjon* R skal kunne kalles en *ordning*, må den minst være både *refleksiv* og *transitiv*. I så fall sier vi at a «er mindre enn eller lik» b istedenfor at a er relatert til b og vi skriver $a \leq b$, og hvis a er relatert til b og b ikke er relatert til a (det betyr spesielt at a og b er forskjellige), sier vi at a er «mindre enn» b og vi skriver $a < b$. Her bruker vi utrykkene «mindre enn» og «mindre enn eller lik» og ulikhetstegnene $\leq$ og $<$ i en overført betydning.

Uttrykkene «større enn» og «større enn eller lik» og ulikhetstegnene $>$ og $\geq$ defineres på vanlig måte. Dvs. $a > b$ er det samme som $b < a$ og $a \geq b$ er det samme som $b \leq a$.

1. *Preordning* En relasjon R på en mengde A kalles en *preordning* hvis den er refleksiv og transitiv. En slik ordning kalles også en *kvasiordning*.
2. *Delvis ordning* En relasjon R på en mengde A kalles en *delvis ordning* hvis den er refleksiv, transitiv og antisymmetrisk. Dette kalles også en *partiell ordning*.
3. *Total preordning* En relasjon R på en mengde A kalles en *total preordning* hvis den er refleksiv, transitiv og total. Kalles også en *total kvasiordning*.
4. *Total ordning* En relasjon R kalles en *total ordning* hvis den er en delvis ordning som også er total. Kalles også en *lineær ordning* eller en *kjede* (eng: chain).

Preordning er den enkleste formen for ordning. Kravet er kun at den tilhørende relasjonen R er refleksiv og transitiv. Det betyr imidlertid at vi kan ha $a \le b$ og $b \le a$ uten at a og b er like. Slike elementer kalles ekvivalente. La S være lik $R \cap R^{-1}$, dvs. lik mengden av par (a, b) slik at $a \le b$ og $b \le a$. S blir refleksiv, transitiv og symmetrisk, dvs. en ekvivalensrelasjon. Ekvivalensklassen $S(a)$ til a er gitt ved $S(a) = \{ b \mid a \le b \text{ og } b \le a \}$. Legg merke til at a alltid er med i $S(a)$ siden R er refleksiv. Hvis vi lar disse ekvivalensklassene være enhetene, vil relasjonen R definere en delvis ordning på dem. Men elementer innen en og samme ekvivalensklasse, dvs. ekvivalente elementer, blir ikke ordnet av R. Se *Oppgave* 1.

I en komparator for en preordning må *compare(a,b)* gi en veldefinert returverdi for alle valg av a og b. Med andre ord må preordningen være total. Her følger to eksempler på slike:

Eksempel 1 Definér følgende relasjon på mengden A av alle tegnstrenger: Tegnstrengen s er relatert til tegnstrengen t hvis de er like eller s kommer foran t alfabetisk uavhengig av bokstavstørrelse (eng: case insensitive order). Det gir en total preordning på A. Alle de som er lik s (uavhengig av bokstavstørrelse) utgjør ekvivalensklassen til s. Hvis f.eks. s = "ABC", vil ekvivalensklassen til s bestå av "ABC", "ABc", "AbC", "Abc", "aBC", "aBc", "abC" og "abc". Komparatoren CASE_INSENSITIVE_ORDER i klassen *String* ordner strenger på denne måten.

Eksempel 2 Definér følgende relasjon på mengden A av alle tegnstrenger: s er relatert til t hvis s har like mange eller færre tegn enn t. Det gir også en total preordning på A. Alle tegnstrenger med samme lengde hører til samme ekvivalensklasse. Det betyr at to strenger med forskjellige lengder har en veldefinert orden, mens strenger med samme lengde ikke har noen bestemt orden. En komparator for dette er laget i *Programkode* 1.4.6 `i`.

En kombinasjon av to preordninger La R og S være to preordninger på samme mengde A. Da kan vi først ordne ekvivalensklassene (definert av ekvivalensrelasjonen $R \cap R^{-1}$) ved hjelp av R og så ordne innholdet i hver ekvivalensklasse ved hjelp av S.

Eksempel 3 I *Programkode* 1.4.8 *c)* ble det laget en generisk teknikk for å kombinere eller sette sammen to komparatorer for samme datatype. Typen ble først ordnet av den ene komparatoren. Hvis det gav 0 som resultat, ble typen ordnet av den andre komparatoren. Det er nettopp slik som beskrivelsen over sier at preordninger kan settes sammen.

La symbolet $S \circ R$ (leses *S ring R*) betegne sammensetningen av preordningene R og S, dvs. R først og så S. Det gir en preordning på A som mengdeteoretisk kan defineres slik:

$$(1.4.11.1) \quad S \circ R \;=\; (R - R^{-1}) \cup (R \cap R^{-1} \cap S)$$

Det står kun som påstand at $S \circ R$ er en preordning. Først må det vises at $S \circ R$ er refleksiv. La $a \in A$. Da er (a, a) med i både R og i S siden begge er refleksive. Men (a, a) er også med i R^{-1} og dermed med i $R \cap R^{-1} \cap S$. Formel (1.4.11.1) gir så at (a, a) er med i $S \circ R$.

Er $S \circ R$ transitiv? La (a, b) og (b, c) være med i $S \circ R$. Da blir det fire tilfeller:

1. $(a, b) \in R$ og $(b, c) \in R - R^{-1}$
2. $(a, b) \in R - R^{-1}$ og $(b, c) \in R \cap R^{-1} \cap S$
3. $(a, b) \in R \cap R^{-1} \cap S$ og $(b, c) \in R - R^{-1}$
4. $(a, b) \in R \cap R^{-1} \cap S$ og $(b, c) \in R \cap R^{-1} \cap S$

Tilfelle 1: Da er både (a, b) og (b, c) med i R og dermed $(a, c) \in R$ siden R er transitiv. Anta at $(a, c) \in R^{-1}$, dvs. at $(c, a) \in R$. Da må også $(b, a) \in R$ siden $(b, c) \in R$ og R er transitiv.

Men (a, b) er ikke med i R^{-1} og dermed er (b, a) ikke med i R. En selvmotsigelse. Altså er (a, c) ikke med i R^{-1}. Konklusjon: (a, c) er med i $R - R^{-1}$ som igjen er inneholdt i $S \circ R$.

Tilfelle 2 vises omtrent på samme måten som tilfelle 1.

Tilfelle 3: Da er (a, c) med i R siden både (a, b) og (b, c) er i R. Anta at (a, c) er med i R^{-1}, dvs. at (c, a) er med i R. Men da må (c, b) være med i R siden (a, b) er i R. Men (c, b) kan ikke være med i R siden (b, c) ikke er med i R^{-1}. Konklusjon: (a, c) er med i $R - R^{-1}$ som igjen er inneholdt i $S \circ R$.

Tilfelle 4: Da er (a, b) og (b, c) med i både R og S. Dermed (a, c) med i $R \cap S$. Både (b, a) og (c, b) er med i R siden (a, b) og (b, c) er i R^{-1}. Dermed er (c, a) med i R og dermed (a, c) med i R^{-1}. Tilsammen gir det at (a, c) er med i $R \cap R^{-1} \cap S$ som igjen er inneholdt i $S \circ R$.

Konklusjonen blir at hvis både (a, b) og (b, c) er med i $S \circ R$, så er (a, c) med i $S \circ R$. Altså er $S \circ R$ transitiv og dermed en preordning.

De tre mengdene R, R^{-1} og $R \cap R^{-1}$ er innbyrdes disjunkte. I tillegg har vi at $(S \circ R)^{-1}$ er lik $S^{-1} \circ R^{-1}$. Dermed gjelder følgende tre formeler der $S \circ R$ og $(S \circ R)^{-1}$ inngår:

(1.4.11.2) $(S \circ R) \cup (S \circ R)^{-1} = (R - R^{-1}) \cup (R^{-1} - R) \cup R \cap R^{-1} \cap (S \cup S^{-1})$

(1.4.11.3) $(S \circ R) \cap (S \circ R)^{-1} = R \cap R^{-1} \cap S \cap S^{-1}$

(1.4.11.4) $(S \circ R) - (S \circ R)^{-1} = (R - R^{-1}) \cup R \cap R^{-1} \cap (S - S^{-1})$

Anta at både R og S er totale preordninger, dvs. at $R \cup R^{-1} = A \times A$ og $S \cup S^{-1} = A \times A$. Da gir formel (1.4.11.2) at $(S \circ R) \cup (S \circ R)^{-1} = (R - R^{-1}) \cup (R^{-1} - R) \cup (R \cap R^{-1}) = R \cup R^{-1} = A \times A$. Det medfører at også $S \circ R$ er en total preordning.

Anta at S er antisymmetrisk, dvs. at $S \cap S^{-1}$ er lik diagonalen D i $A \times A$. Da ser vi av formel (1.4.11.3) at også $S \circ R$ er antisymmetrisk. Det betyr spesielt at hvis R er en preordning og S en delvis ordning, så blir også $S \circ R$ en delvis ordning. Hvis R er en delvis ordning, vil derimot $S \circ R = R$, dvs. S har ingen effekt.

Sammensetningen $S \circ R$ er ikke kommutativ, dvs. vi har generelt ikke at $S \circ R = R \circ S$ siden $(R - R^{-1}) \cup (R \cap R^{-1} \cap S)$ normalt er forskjellig fra $(S - S^{-1}) \cup (S \cap S^{-1} \cap R)$.

Vi har imidlertid at sammensetningen er assosiativ, dvs. at hvis R, S og T er preordninger, så er $T \circ (S \circ R) = (T \circ S) \circ R$ Det vises slik ved hjelp av (1.4.11.3) og (1.4.11.4):

$$
\begin{aligned}
& T \circ (S \circ R) \\
&= ((S \circ R) - (S \circ R)^{-1}) \cup (S \circ R) \cap (S \circ R)^{-1} \cap T \\
&= (R - R^{-1}) \cup R \cap R^{-1} \cap (S - S^{-1}) \cup R \cap R^{-1} \cap S \cap S^{-1} \cap T \\
&= (R - R^{-1}) \cup R \cap R^{-1} \cap ((S - S^{-1}) \cup (S \cap S^{-1} \cap T)) \\
&= (T \circ S) \circ R
\end{aligned}
$$

Komparatorer for preordninger La A være mengden av alle instanser av en datatype og anta at vi har en preordning for A. En komparator for denne preordningen må være slik at for hvert valg av instanser a og b må *compare(a,b)* returnere en veldefinert verdi eller kaste et unntak hvis a og b ikke kan sammenlignes, dvs. at hverken $a \leq b$ eller $b \leq a$. Her tar vi

imidlertid som gitt at preordningen er total, dvs. at vi alltid har $a \leq b$ eller $b \leq a$. Som nevnt tidligere sier vi at a og b er *ekvivalente* hvis $a \leq b$ og $b \leq a$. Vi skal her bruke $a \approx b$ som betegnelse for at a og b er ekvivalente. Dermed vil en og bare en av flg. muligheter inntreffe: 1) $a < b$, 2) $a \approx b$ eller 3) $a > b$. Metoden *compare* må derfor kodes slik at

1. returverdien er negativ hvis og bare hvis $a < b$
2. returverdien er 0 hvis og bare hvis $a \approx b$
3. returverdien er positiv hvis og bare hvis $a > b$

I en preordning kan to elementer a og b være ekvivalente og dermed at *compare(a,b)* vil returnere 0, uten at de er like. Vi har at $a < b$ er det samme som $b > a$ og at $a \approx b$ er det samme som $b \approx a$. For at komparatoren skal bli konsistent må vi ha at hvis *compare(a,b)* er negativ, så må *compare(b,a)* være positiv og omvendt. Tilsvarende hvis *compare(a,b)* = 0, må *compare(b,a)* = 0 og omvendt. Det er ikke noe krav, men normalt vil man forsøke å kode det slik at vi alltid har *compare(a,b)* = − *compare(b,a)*.

Delvise og totale ordninger Det som er mest vanlig er delvise og totale ordninger. Det er en delvis ordning hvis den tilhørende relasjonen R er *refleksiv*, *transitiv* og *antisymmetrisk*, og det er en total ordning hvis den er *refleksiv*, *transitiv*, *antisymmetrisk* og *total*.

Eksempel 4 La mengden A bestå av alle hele tall. Definer flg. relasjon på A: Et tall a er relatert til et tall b hvis og bare hvis a er mindre enn eller lik b, dvs. $a \leq b$. Dette gir en total ordning av A og er selvfølgelig det samme som den vanlige naturlige ordningen av tallene i A.

Eksempel 5 La S være mengden av alle tegnstrenger. Definer flg. relasjon på S: En streng s er relatert til en streng t hvis og bare hvis s er lik eller kommer foran t alfabetisk. Dette gir en total ordning av S og er selvfølgelig den vanlige alfabetiske ordningen.

Eksempel 6 La $A = \{2,3,4,6,8,9,10,12\}$ og la R være mengden av par (a, b) slik at a går opp i b. Et positivt tall går alltid opp i seg selv, dvs. R er refleksiv. Hvis a er forskjellig fra b og a går opp i b, så går ikke b opp i a. Dermed er R antisymmetrisk. Hvis a går opp i b og b går opp i c, så går a opp i c. Det gir at R er transitiv. Med andre ord definerer R en delvis ordning på A. La $a = 4$ og $b = 6$. Vi vet at 4 ikke går opp i 6 og at 6 ikke går opp i 4. Det betyr at a og b ikke er sammenlignbare. Ordningen er derfor ikke total.

Eksempel 7 Elementene i en mengde A med en total ordning har en bestemt rekkefølge med hensyn på ordningen. Det er også mulig å gå den omvendte veien. Vi kan bestemme en rekkefølge for elementene i A og så bruke rekkefølgen til å definere en total ordning. Gitt elementene a og b. Da sier vi at $a \leq b$ hvis $a = b$ eller a kommer foran b i rekkefølgen. La for eksempel A være heltallene fra 1 til 10. Da vil rekkefølgen $7, 3, 5, 1, 6, 10, 2, 9, 4, 8$ definere en total ordning. Det betyr f.eks. at $5 \leq 2$ siden 5 kommer foran 2 i rekkefølgen. Dette er selvfølgelig en kunstig ordning, men den oppfyller alle kravene til en total ordning.

Eksempel 8 La A bestå av flg. punkter i et x,y-koordinatsystem: (5,1), (7,2), (6,3), (7,4), (6,5), (3,3), (4,4), (5,6), (1,2) og (2,5). Denne rekkefølgen bestemmer en total ordning. F.eks. er (5,1) ≤ (3,3) siden (5,1) kommer foran (3,3) i rekkefølgen. Dette ser kunstig ut, men er egentlig en viktig ordning for punkter i f.eks. 1. kvadrant i et x,y-koordinatsystem. Hvis en tegner den rette linjen fra origo til (5,1) og den rette linjen fra origo til (3,3), vil en se at den første linjen danner en mindre vinkel med x-aksen enn den andre linjen. Generelt kan vi definere flg. ordning for punkter: Et punkt p er «mindre enn» et punkt q hvis den rette linjen fra origo til p danner en mindre vinkel med x-aksen enn den rette linjen fra origo til q. Hvis de to linjene gir samme vinkel, så er p «mindre enn» q hvis p ligger nærmere origo enn q. Tegn de 10 punktene i et x,y-koordinatsystem og sjekk at den rekkefølgen de 10 punktene er satt opp i, gir den oppgitte «mindre enn»-regelen. Se *Oppgave 3*.

Koding av totale ordninger Hvis en total ordning for instanser av en datatype X er naturlig, skal X normalt gjøres sammenlignbar med seg selv (eng: selfcomparable), dvs. X skal implementere *Comparable<X>*. Spørsmålet er selvfølgelig hva som menes med en naturlig ordning. Det er ingen bestemte kriterier for det. Men ordningen må i hvert fall være total. Hvis den total ordningen ikke er naturlig, er løsningen vanligvis å lage en komparator.

Hvis x og y er to instanser av X og X har en total ordning, gjelder et og bare et av flg. tre tilfeller: 1) $x < y$, 2) $x = y$ eller 3) $x > y$. Sammenligningsmetoden (*compareTo* eller *compare*) må derfor kodes slik at

1. returverdien er negativ hvis $x < y$
2. returverdien er 0 hvis og bare hvis $x = y$
3. returverdien er positiv hvis $x > y$

Vi har at $x < y$ er det samme som $y > x$ og at $x = y$ er det samme som $y = x$. For at det skal bli konsistent må vi være nøye med kodingen. La *compare* være sammenligningsmetoden. (Det blir helt tilsvarende for *compareTo*.) Den må kodes slik at hvis *compare(x , y)* er negativ, må *compare(y , x)* være positiv, og omvendt. Tilsvarende hvis *compare(x , y)* = 0, må *compare(y , x)* = 0, og omvendt. Det er ikke noe krav, men normalt vil man forsøke å kode det slik at vi alltid har *compare(x , y)* = − *compare(y , x)*.

Leksikografiske ordninger La X være en mengde med en ordning. Vi sier at to elementer x_1 og x_2 i X er ekvivalente hvis $x_1 \le x_2$ og $x_2 \le x_1$. Vi betegner det med $x_1 \approx x_2$. Husk at hvis det er en preordning, så kan elementene x_1 og x_2 være forskjellige selv om $x_1 \approx x_2$. Men hvis det er en delvis ordning eller en total ordning, så er $x_1 \approx x_2$ det samme som $x_1 = x_2$.

La nå A og B være to mengder som har hver sin ordning. Da definerer vi en relasjon $\le$ på produktmengden $A \times B$ ved at hvis (a_1 , b_1) og (a_2 , b_2) er to elementer i $A \times B$, så er

(1.4.11.5) $\quad (a_1 , b_1) \le (a_2 , b_2)$ hvis $a_1 < a_2$ eller $a_1 \approx a_2$ og $b_1 \le b_2$.

Ulikheten $a_1 < a_2$ og ekvivalensen $a_1 \approx a_2$ er mhp. ordningen av A og ulikheten $b_1 \le b_2$ er mhp. ordningen av B. Relasjonen (1.4.11.5) er en *leksikografisk* ordning av $A \times B$. Vi har flg. sammenheng mellom ordningene på A og B og på den leksikografiske ordningen av $A \times B$:

1. den er en preordning hvis ordningene på A og B er preordninger
2. den er en total preordning hvis ordningene på A og B er totale preordninger
3. den er en delvis ordning hvis ordningene på A og B er delvise ordninger
4. den er en total ordning hvis ordningene på A og B er totale ordninger

Her nøyer vi oss med å vise at tilfelle 1. stemmer. De tre andre tilfellene kan vises på en tilsvarende måte. La (a , b) være med i $A \times B$. Vi har at $a \le a$ og $b \le b$ sidene ordningene på både A og B er refleksive. Men vi har også at $a \approx a$, men ikke at $a < a$. I henhold til (1.4.11.5) betyr det at $(a , b) \le (a , b)$, dvs. den leksikografiske ordningen er refleksiv.

Anta så at $(a_1 , b_1) \le (a_2 , b_2)$ og at $(a_2 , b_2) \le (a_3 , b_3)$. Dette gir fire muligheter:

1. $a_1 < a_2$, $a_2 < a_3$
2. a_1 a_2 , $a_2 \approx a_3$ og $b_2 \le b_3$
3. $a_1 \approx a_2$ og $b_1 \le b_2$, $a_2 < a_3$
4. $a_1 \approx a_2$ og $b_1 \le b_2$, $a_2 \approx a_3$ og $b_2 \le b_3$

1. Vi får $a_1 < a_3$ siden ordningen på A er transitiv og dermed at $(a_1 , b_1) \le (a_3 , b_3)$.
2. Hvis $a_1 < a_2$ og $a_2 \approx a_3$, så blir a_1 a_3 og dermed $(a_1 , b_1) \le (a_3 , b_3)$.

3. Hvis $a_1 \approx a_2$ og $a_2 < a_3$, så blir $a_1 < a_3$ og dermed $(a_1, b_1) \leq (a_3, b_3)$.
4. Hvis $a_1 \approx a_2$ og $a_2 \approx a_3$, så blir $a_1 \approx a_3$. Videre vil $b_1 \leq b_2$ og $b_2 \leq b_3$ gi at $b_1 \leq b_3$. Dermed $(a_1, b_1) \leq (a_3, b_3)$.

Tilsammen får vi at den leksikografiske ordningen er transitiv.

Eksempel 9 La *A* være mengden av alle etternavn og *B* mengden av alle fornavn. Både *A* og *B* er alfabetisk ordnet, dvs. begge har en total ordning. Klassen *Person* i *Avsnitt* 1.4.4 har etternavn og fornavn som instansvariabler og den er ordnet på samme måte som den leksikografiske ordningen av produktmengden $A \times B$. Det betyr at det blir en total ordning.

Eksempel 10 I *Avsnitt* 1.4.5 ble klassen *Student* definert som en subklasse til klassen *Person*. En student har dermed fornavn, etternavn og klasse. La *A* være mengden av alle klasser, *B* mengden av alle etternavn og *C* mengden av alle fornavn. Komparatoren (laget ved hjelp av et lamda-uttryk) i *Programkode* 1.4.6 *d)* er laget slik at ordningen svarer til den leksikografiske ordningen av $A \times B \times C$.

Eksempel 11 I *Avsnitt* 1.4.9 ble klassen *Dato* definert med *år*, *mnd* og *dag* som instansvariabler. La *A* være mengden av år, *B* mengden av måneder (1 - 12) og *C* mengden av dager (1 - 28/29/30/31). Hver av disse kan ordnes på vanlig måte som tallmengder. Den naturlige måten å ordne datoer er å først å ordne etter år, så etter måned og til slutt etter dag. Dette svarer nøyaktig til den leksikografiske ordningen av $A \times B \times C$.

Koding av leksikografiske ordninger Anta at vi har en leksikografisk ordning av $A \times B$. Da benyttes ordningen av *A* først. Hvis den ikke gir likhet (eller ekvivalens hvis det er en preordning), så returneres resultatet. Hvis de gir likhet (eller ekvivalens), returneres isteden det som ordningen av *B* gir. Vi kan bruke klassen *Person* som eksempel. Da er *A* mengden av etternavn og *B* mengden av fornavn. Metoden *compareTo* i klassen *Person* er laget slik:

```
public int compareTo(Person p)
{
  int cmp = etternavn.compareTo(p.etternavn);
  if (cmp != 0) return cmp;
  return fornavn.compareTo(p.fornavn);
}
```

Programkode* 1.4.11 *a)

Mhp. ordningen over vil *Kari Olsen*, *KARI Olsen* og *KARI OLSEN* være forskjellige personer. Vi kan isteden bruke en preordning på både etter- og fornavn. Dvs. preordningen som ikke tar hensyn til bokstavstørrelse. Dette vil gi en leksikografisk ordning av personer der de tre personene blir sett på som ekvivalente. En komparator som gjør dette kan lages slik:

```
Comparator<Person> c = (p1, p2) ->
{
  int cmp = p1.etternavn().compareToIgnoreCase(p2.etternavn());
  return cmp != 0 ? cmp : p1.fornavn().compareToIgnoreCase(p2.fornavn());
};
```

eller slik:

```
Comparator<Person> c =
Comparator.comparing((Person p) -> p.etternavn().toUpperCase()).
thenComparing(p -> p.fornavn().toUpperCase());
```

Programkode* 1.4.11 *b)

Oppgaver til Avsnitt 1.4.11

1. Vis at hvis en relasjon R på en mengde A er refleksiv og transitiv, så vil relasjonen S på A definert ved $S = R \cap R^{-1}$, være en ekvivalensrelasjon.
2. Gitt preordningen i *Eksempel* 1. Hvor mange tengstrenger vil ekvivalensklassen til strengen "ABCD" inneholde? Hvor mange vil ekvivalensklasen til "algoritme" inneholde?
3. Lag en tegning som viser de 10 punktene i *Eksempel* 8 lagt inn i et *x,y*-koordinatsystem. Tegn rette linjer fra origo til hvert av punktene. Sjekk så at den rekkefølgen punktene er satt opp i svarer til den ordningen som er beskrevet i eksempelet.
4. Et heltall av typen *int* kan ses på som en bitsekvens med 32 biter. Bitsekvenser kan sammenlignes leksikografisk. Hvis to bitsekvenser er ulike, sammenligner vi bitene i den første posisjonen fra venstre hvor de er ulike. Da regnes 0 som mindre enn 1. Lag klassen *BitSekvens*. Den skal kun ha en *int* som instansvariabel. La standardkonstruktøren sette variablen til 0. Lag i tillegg en konstruktør som gir instansvariablen en vilkårlig *int*-verdi. Klassen skal implementere *Comparable<BitSekvens>*. Implementer også metodene *equals*, *toString* og *HashCode*. Metoden *toString* skal lage en tegnstreng som inneholder alle de 32 bitene.
5. På HiOA er hver datastudent registrert i en klasse. For eksempel står klassebetegnelsen 2IA for klasse *A* på 2. årstrinn på IT-studiet. Men det er ingen fast regel om det skal skrives som 2ia, 2IA, 2Ia eller 2iA. Gjør om komparatoren (bruk `Comparator` istedenfor `Komparator`) i *Programkode* 1.4.6 *h)* slik at alle slike varianter i skrivemåten behandles som samme klassekode.

1.4.12 Ordninger, *equals* og *hashCode*

Hvis vår klasse *X* skal implementere `Comparable<X>`, må det skje på basis av en total ordning. La *x* og *y* være to instanser av klassen *X*. Da må koden for `compareTo`-metoden lages slik at:

1. Hvis *x* er «mindre enn» *y*, blir `x.compareTo(y) < 0`.
2. Hvis *x* er «lik» *y*, blir `x.compareTo(y) = 0`.
3. Hvis *x* er «større enn» *y*, blir `x.compareTo(y) > 0`.
4. Hvis `x.compareTo(y) = 0`, blir *x* «lik» *y*.

La *x* være forskjellig fra *y*. Vi har at *x* «mindre enn» *y* er det samme som *y* «større enn» *x*. Da følger det av reglene at `x.compareTo(y)` og `y.compareTo(x)` må ha motsatt fortegn. Videre, hvis de er like, må både `x.compareTo(y)` og `y.compareTo(x)` være 0. Det beste er å kode metoden slik at vi får `x.compareTo(y) = –y.compareTo(x)`.

Når en klasse lages sammenlignbar, må `equals`-metoden kodes slik at den blir konsistent med `compareTo()`, dvs. `x.compareTo(y) = 0` hvis og bare hvis `x.equals(y) = true`.

Alle klasser arver metoden `equals` fra basisklassen `Object`. To instanser *x* og *y* sies å være like hvis `x.equals(y) = y.equals(x) = true`. Hvis vi ikke overstyrer (eng: override) `equals`-metoden i vår klasse, får vi den versjonen som er kodet i basisklassen. Den virker slik at to instanser *x* og *y* sies å være like hvis de refererer til det samme «fysiske» objektet. Dette vil i en del tilfeller være ok. Men hvis to instanser *x* og *y* anses som «logisk» like selv om de representerer to (fysisk) forskjellige objekter, så fungerer ikke den arvede metoden.

Eksempel 1: I klassen `Person` er `equals`-metoden kodet. I din versjon av klassen `Person` har du kanskje enda en `equals`-metode (se *Oppgave* `2f` i *Avsnitt* 1.4.4). Den har en `Person` som argument. Mer om det senere. Kommenter vekk metoden `equals()` (begge to hvis du har to av dem). Da vil flg. kodebit likevel virke, men gi et annet resultat enn det en ville forvente:

```
Person p = new Person("Anne","Olsen");     // Anne Olsen
Person q = new Person("Anne","Olsen");     // Anne Olsen på nytt

System.out.print(p.compareTo(q) + " " + p.equals(q));  // Utskrift: 0 false
```

Programkode* 1.4.12 *a)

Hvis vi ikke koder `equals`-metoden i klassen `Person`, vil den arvede metoden se på *p* og *q* i *Programkode* 1.4.12 *a)* som to forskjellige objekter og dermed som to forskjellige personer. Men for oss representerer de to samme person. Derfor må vi kode `equals()` (den med `Object` som argument) og kode den slik at den gir `true` hvis og bare hvis `compareTo()` gir 0.

Hvis vi har kodet `compareTo` først, er det lett å lage `equals` ved hjelp av den. Det finnes en standard måte å gjøre det på. La vår klasse for eksempel hete `X`. Da kan metoden lages slik:

```
public boolean equals(Object o)          // må ha Object som parametertype
{
  if (o == this) return true;            // er det samme objekt?
  if (!(o instanceof X)) return false;   // er det rett klasse?
  return compareTo((X)o) == 0;           // bruker klassens compareTo
}
```

Programkode* 1.4.12 *b)

Poenget er å overstyre den arvede metoden. Derfor må paramtertypen være `Object` siden den er det i klassen `Object`. I første setning sjekkes det om vi sammenligner et objekt med

seg selv. Hvis det skjer ofte at instanser av klassen X sammenlignes med seg selv (to referanser til samme objekt), er dette fordelaktig. Det er imidlertid fullt mulig å ta vekk setningen siden compareTo skal gi 0 når to referanser til samme objekt sammenlignes. Det må også sjekkes at parameterobjektet *o* har rett type. Det kan f.eks. gjøres ved hjelp av operatoren instanceof eller ved hjelp av metoden getClass(). Se *Oppgave* 1.

Hvis *o* er av feil type, skal metoden returnere false. (Hvis *o* er null, er det av feil type). Hvis rett type, må vi først konvertere siden compareTo-metoden krever at parametertypen er X. Hvis compareTo er kodet slik at to instanser er like hvis og bare hvis den returnerer 0, så vil equals-metoden bli korrekt.

I noen situasjoner kan det lønne seg å kode equals-metoden på en mer direkte måte. Den vil da kunne bli noe mer effektiv. I flg. eksempel er equals() for klassen Person i *Avsnitt* 1.4.4 kodet uten at vi sammenligner med this og bruker compareTo-metoden:

```
public boolean equals(Object o)      // ny versjon av equals
{
  if (!(o instanceof Person)) return false;
  final Person p = (Person)o;
  return etternavn.equals(p.etternavn) && fornavn.equals(p.fornavn);
}
```

Programkode* 1.4.12 *c)

En fordel med implementasjonen i *Programkode* 1.4.12 *b)* er at hvis metoden compareTo() er korrekt laget, så vil x.equals(y) og y.equals(x) alltid gi samme svar og to instanser *x* og *y* vil være like hvis og bare hvis x.equals(y) er true. Hvis vi koder equals-metoden direkte slik som i *Programkode* 1.4.12 *c)*, må vi selv passe på at metoden oppfører seg riktig.

Eksempel 2: I *Avsnitt* 1.4.5 ble klassen Student laget som en subklasse til Person. Men hverken compareTo() eller equals() ble overstyrt, dvs. det ble ikke laget egne versjoner av dem i klassen Student. Flg. kode vil likevel virke siden en Student er en Person:

```
Student s = new Student("Anne","Olsen",Studium.IT);
Student t = new Student("Anne","Olsen",Studium.IT);

System.out.println(s.compareTo(t) + " " + s.equals(t));  // 0 true
```

Programkode* 1.4.12 *d)

Klassen Person inneholdt opprinnelig kun én equals-metode, dvs. den som har Object som parametertype. Den kan også ha en metode med den tilhørende klassen som parametertype. Den er enklere å lage siden en ikke trenger å sjekke datatypen. Den kan lages slik:

```
public boolean equals(Person p)         // Person som parametertype
{
  if (p == null) return false;          // null-argument
  return etternavn.equals(p.etternavn) && fornavn.equals(p.fornavn);
}
```

Programkode* 1.4.12 *e)

Det holder imidlertid ikke bare med equals-metoden fra *Programkode* 1.4.12 *e)* i klassen vår. Den blir brukt når parameteren er av typen Person, men ikke ellers. Vi gjør et eksperiment: Kommenter vekk den equals-metoden i Person som har Object som parametertype. Den andre skal være der, dvs. den fra *Programkode* 1.4.12 *e)* over. Dermed har vi helt sikkert en equals-metode som kan sammenligne personer. Kjør så flg. kodebit:

```
List<Person> l = new ArrayList<>();          // oppretter en liste
l.add(new Person("Anne","Olsen"));           // Anne Olsen legges inn
System.out.println(l);                       // Utskrift: [Anne Olsen]

boolean søk = l.contains(new Person("Anne","Olsen"));    // søker
System.out.println(søk);   // Utskrift:  false
```

Programkode* 1.4.12 *g)

I *Programkode* 1.4.12 *g)* legges Anne Olsen inn i en liste. Vi ser at hun er der siden hun blir skrevet ut. Men søkingen gir `false` som resultat. Hva har skjedd? Metoden `equals()` kalles implisitt i `contains()` og siden `contains()` har `Object` som parametertype, vil kompilatoren lete etter en `equals`-metode med samme type. Men den er midlertidig fjernet. Da henter kompilatoren `equals`-versjonen vi arver fra klassen `Object`. Men den sier at to objekter er like kun hvis det er samme objekt. Vi har laget to forskjellige objekter (new to ganger) som begge har Anne Olsen som innhold. Men de ses på som ulike. Derfor gir `contains` false.

Hvis vi derimot oppretter Anne Olsen slik: `Person p = new Person("Anne","Olsen");` og så bruker *p* som argument både i `add()` og i `contains()`, vil søket gi `true`. Se *Oppgave* 2 og 3.

Det er også mulig å sammenligne ved hjelp av operatoren `==`. Det er den vi bruker når vi sammenligner instanser av standardtypene (`int`, `long`, `char`, osv.) I flg. eksempel sjekkes om et heltall av typen `int` er partall eller oddetall:

```
int n = 123;
String type = (n % 2 == 0) ? "partall" : "oddetall";
System.out.println(n + " er " + type);
```

Programkode* 1.4.12 *h)

Med objektreferanser blir det annerledes. En referanse er adressen til det stedet i minnet der objektet ligger. I slike tilfeller sammenligner operatoren == adressene. Se på flg. eksempel:

```
Person p = new Person("Anne","Olsen");    // p er Anne Olsen
Person q = p;                             // q er Anne Olsen
Person r = new Person("Anne","Olsen");    // r er Anne Olsen

System.out.println((p == q) + " " + (p == r));  // true false
```

Programkode* 1.4.12 *i)

Sammenligningen `p == q` i *Programkode* 1.4.12 *i)* gir `true` siden *p* og *q* begge refererer til det samme objektet, dvs. samme adresse. Men `p == r` gir `false` siden *p* og *r* ikke refererer til samme objekt. De to objektene har samme innhold, men ligger på forskjellige steder i minnet. Hver gang vi bruker operatoren `new` får vi et nytt objekt.

Det er imidlertid situasjoner der en kan lure på hva som egentlig skjer. Se flg. eksempel:

```
String p = "Anne Olsen";
String q = "Anne Olsen";
String r = new String("Anne Olsen");

System.out.println((p == q) + " " + (p == r));        // true false
System.out.println(p.equals(q) + " " + p.equals(r));  // true true
```

Programkode* 1.4.12 *j)

`"Anne Olsen"` kalles en strengkonstant (eng: string literal). Første gang en strengkonstant brukes, vil det bli opprettet et objekt med innholdet av konstanten som innhold. Med andre ord vil *p* referere til et `String`-objekt. «Bak kulissene» er det imidlertid organisert slik at kompilatoren vil oppdage om en tidliger strengkonstant brukes på nytt. Dermed vil *q* referere til det samme objektet som *p* og `p == q` blir `true`. Men *r* derimot, vil referere til et helt nytt objekt siden vi bruker `new`. Derfor blir `p == r` lik `false`. I metoden `equals()` i klassen `String` sammenlignes innholdet av objektene. Derfor blir det `true` i begge tilfellene. Sjekk hvordan `equals()` er kodet i klassene `String` og `Object`. Se *Oppgave* 4, 5, 6.

Huskeregel:

- Brukes operatoren == på objektreferanser, sammenlignes referansene som adresser.
- Hvis *`equals`*`()` ikke er kodet og likevel brukes, vil den virke som operatoren == .
- Hvis *`equals`*`()` er kodet (og kodet korrekt), vil den sammenligne innholdet av de objektene som referansene går til.
- En nybegynnerfeil (og en feil selv erfarne programmerere kan gjøre) er å bruke == der en skal bruke *`equals`*`()`. Det gir ingen syntaksfeil, men en kan få gale resultater.

Alle klasser arver metoden *`hashCode`*`()` fra basisklassen `Object`. Den returnerer et stort heltall som vanligvis er konstruert ved hjelp objektets minneadresse. Hvis det er aktuelt å bruke en hashing-teknikk, må `equals()` og *`hashCode`*`()` være konsistente. Hvis *`equals`*-metoden sier at *x* og *y* er like, så må `x.`*`hashCode`*`() = y.`*`hashCode`*`()`. Mer om dette i *Kapittel 7*.

Oppgaver til Avsnitt 1.4.12

1. Bytt ut koden: `if (!(o instanceof Person)) return false;` i *Programkode 1.4.12 c)* med: `if (getClass() != o.getClass()) return false;` Sjekk at det virker på samme måte. Vi må da imidlertid ha med: `if (o == null) return false;` foran. Det slipper vi i det første tilfellet siden `instanceof` gir `false` hvis *o* er `null`.
2. Kommenter vekk metoden `equals(Object o)` fra klassen `Person`, men den i *Programkode 1.4.12 e)* skal være der. Kjør så programmet i *Programkode 1.4.12 g)*. Sjekk utskriften. Ta så metoden tilbake. Kjør programmet igjen. Hva blir nå utskriften?
3. Gjør som i *Oppgave* 1, dvs. kommenter vekk metoden `equals(Object o)` fra klassen `Person`, men la den i *Programkode 1.4.12 e)* være der. Opprett så en `Person` eksplisitt i *Programkode 1.4.12 g)*, dvs. ved: `Person p = new Person("Anne","Olsen");` Bruk så *p* i metodene `add()` og `contains()`. Hva blir nå utskriften når programbiten kjøres?
4. I standardoppsettet av NetBeans kommer en advarsel hvis objektreferanser sammenlignes ved hjelp av operatoren ==. Sjekk at det stemmer. Det er fort gjort å glemme seg bort. Som oftest er det `equals()` en skulle ha brukt siden den sammenligner innholdet av de objektene det refereres til. Operatoren == sammenligner referansene, dvs. adressene.
5. De samme strengkonstantene kan inngå mange steder. Se flg. kode:
   ```
   Person p = new Person("Anne", "Olsen");
   String s = "Anne", t = "Olsen", u = "Anne Olsen";
   String v = "Anne" + " " + "Olsen";
   String[] navn = {"Anne", "Olsen"};
   ```
   Bruk operatoren == til å sammenligne de referansene du tror er like? Hva med f.eks. `p.etternavn() == navn[1]`? Eller `u == v`? Osv.
6. Sjekk hvordan `equals()` er kodet i klassene `String` og `Object`.

1.4.13 Delvise ordninger

En relasjon R på en mengde A kalles en *delvis ordning* hvis den er *refleksiv*, *transitiv* og *antisymmetrisk* - se *Avsnitt* 1.4.11. Hvis den i tillegg er *total* kalles det en *total ordning*.

Eksempel 1 La $A = \{2, 3, 4, 6, 8, 9, 10, 12\}$ og la R være mengden av par (a, b) slik at a går opp i b. Et positivt tall går alltid opp i seg selv, dvs. R er refleksiv. Hvis a er forskjellig fra b og a går opp i b, så går ikke b opp i a. Dermed er R antisymmetrisk. Hvis a går opp i b og b går opp i c, så går a opp i c. Det gir at R er transitiv. Med andre ord er dette en delvis ordning. La $a = 4$ og $b = 6$. Vi vet at 4 ikke går opp i 6 og at 6 ikke går opp i 4. Med andre ord er verken (a, b) eller (b, a) med i R. Ordningen er derfor ikke total.

Eksempel 2 La mengden A være gitt ved $A = \{1, 2, 3\}$ og la $P(A)$ være mengden av alle delmengder av A. Dvs. $P(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$. Definer flg. relasjon på $P(A)$: En delmengde D_1 av A er relatert til en delmengde D_2 hvis D_1 er inneholdt i D_2. F.eks. er $\{1,2\}$ relatert til $\{1,2,3\}$ siden $\{1,2\}$ er inneholdt i $\{1,2,3\}$. Relasjonen er åpenbart refleksiv, transitiv og antisymmetrisk – med andre ord er det en delvis ordning. La $D_1 = \{1, 2\}$ og $D_2 = \{2, 3\}$. Da er verken D_1 inneholdt i D_2 eller D_2 inneholdt i D_1. Ordningen er med andre ord ikke total.

Det vil være situasjoner der vi trenger en komparator for en delvis ordning. Men hva skal komparatorens `compare`-metode returnere hvis to objekter x og y ikke er sammenlignbare, dvs. hvis vi verken har $x \le y$ eller $y \le x$? Vi bestemmer at den da skal returnere 0.

Anta at komparatoren skal lages for en delvis ordning for en klasse X. La x og y være to instanser. Som vanlig skriver vi $x < y$ hvis $x \le y$ og $x \ne y$. Da skal `compare`-metoden:

1. returnere et negativt heltall hvis $x < y$
2. returnere et positivt heltall hvis $x > y$
3. returnere 0 hvis $x = y$
4. returnere 0 hvis x og x ikke er sammenlignbare

En slik komparator vil ikke bli konsistent med `equals`-metoden. Her vil `compare`-metoden returnere 0 både for like verdier og for verdier som ikke er sammenlignbare.

En komparator for den ordningen som er diskutert i *Eksempel* 1 kan kodes slik:

```
Comparator<Integer> c = (x, y) ->
{
  if (x == y) return 0;                 // x og y er like
  else if (y % x == 0) return -1;       // x går opp i y
  else if (x % y == 0) return 1;        // y går opp i x
  else return 0;                        // ikke sammelignbare
};
```

Programkode* 1.4.13 *a)

Legg merke til at vi kan bruke operatorene % og == selv om datatypen er *Integer* og ikke *int*. Kompilatoren konverterer (autoavboksing) til *int*-format før operasjonene utføres.

Hvis en delvis ordnet mengde har endelig mange elementer, vil det blant disse finnes et eller flere som er *minimale* og et eller flere som er *maksimale*:

> **Definisjon 1.4.13 *a)*** Et element a i en delvis ordnet mengde A kalles *minimalt* hvis det ikke finnes noen elementer som er mindre, dvs. hvis b er et element i A slik at $b \le a$, så må $a = b$. Et element a kalles *maksimalt* hvis det ikke finnes noen som er større, dvs. hvis $b \ge a$, så må $a = b$.

Eksempel 3 I *Eksempel* 1 er tallet 2 et minimalt element siden det ikke er noen andre elementer fra *A* (bortsett fra 2 selv) som går opp 2. Tallet 3 er også et minimalt element siden ingen av de øvrige tallene går opp i 3. Men ingen av de andre er minimale. Både 8, 9, 10 og 12 er maksimale fordi ingen av dem går opp i noen andre tall fra *A* enn seg selv.

Flg. eksempel viser hvordan vi kan bruke *maks*-metoden fra *Programkode* 1.4.6 *g)* og en komparator til finne et maksimalt element i en tallmengde mhp. ordningen i *Eksempel* 1:

```
Integer[] a = {3,8,2,10,9,12,6,4};

Comparator<Integer> c = (x, y) ->
{
  if (x == y) return 0;                 // x og y er like
  else if (y % x == 0) return -1;       // x går opp i y
  else if (x % y == 0) return 1;        // y går opp i x
  else return 0;                        // ikke sammelignbare
};

int ma = Tabell.maks(a, c);             // maksimum
System.out.println(a[ma]);              // utskrift: 9

int mi = Tabell.maks(a, c.reversed());  // minimum
System.out.println(a[mi]);              // utskrift: 3
```

Programkode* 1.4.13 *b)

Mengden {2, 3, 4, 6, 8, 9, 10, 12} har fire maksimale og to minimale elementer mhp. ordningen i *Eksempel* 1. Tabellen i *Programkode* 1.4.13 *b)* inneholder de samme verdiene, men i en litt annen rekkefølge. *Maks*-metoden starter med det første elementet i tabellen, dvs. tallet 3. For hvert tall videre i tabellen sjekkes det om 3 går opp i tallet. Det skjer når vi kommer til 9. Videre derfra sjekkes det om det er noen tall som 9 går opp i. Men det er det ikke. Derfor er 9 et maksimalt element. Metoden returnerer posisjonen til tallet 9. Hvis vi bytter om på rekkefølgen i tabellen, vil *maks*-metoden kunne gi et annet maksimalt element som svar. Hvis vi f.eks. bruker {2, 3, 4, 6, 8, 9, 10, 12} får vi 8 som svar. Se *Oppgave* 1.

Eksempel 4 I *Eksempel* 2 er den tomme mengden Ø (eller {}) et minimalt element. Ingen andre delmengder er inneholdt i Ø. Dermed er Ø et minimalt element og det eneste siden Ø er inneholdt i alle delmengder. Mengden { 1 , 2 , 3 } er det eneste maksimale elementet.

Hvis vi bruker instanser av klassen `Tallmengde` fra *Avsnitt* 1.3.16 som mengder, vil vi ved hjelp av flg. komparator finne det minimale og det maksimale elementet med hensyn på ordningen fra *Eksempel* 2:

```
Comparator<Tallmengde> c = (A, B) ->
{
  if (A.equals(B)) return 0;            // A og B er like
  else if (B.inklusjon(A)) return -1;   // A delmengde av B
  else if (A.inklusjon(B)) return 1;    // B delmengde av A
  else return 0;                        // ikke sammelignbare
};

int[][] A = { {1,2}, {2,3}, {3}, {1,2,3}, {1}, {1,3}, {2}, {} };

Tallmengde[] a = new Tallmengde[A.length];

for (int i = 0; i < A.length; i++) a[i] = new Tallmengde(A[i]);
```

```
Tallmengde Maks = a[Tabell.maks(a,c)];
Tallmengde Min = a[Tabell.maks(a,c.reversed())];

System.out.println(Arrays.toString(a));
System.out.println(Min + " er en minimal mengde");
System.out.println(Maks + " er en maksimal mengde");

// [{1,2}, {2,3}, {3}, {1,2,3}, {1}, {1,3}, {2}, {}]
// {} er en minimal mengde
// {1,2,3} er en maksimal mengde
```

Programkode* 1.4.13 *c)

Toplogisk sortering En sorterering av en totalt ordnet mengde gir en entydig rekkefølge. Det er også mulig å sortere en delvis ordnet mengde, men da på en måte som ikke gir et entydig resultat. Vi kaller det en *topologisk* sortering:

> **Definisjon 1.4.13 *b)*** La A være en delvis ordnet mengde med endelig mange elementer. En *topologisk* sortering av A er en rekkefølge av elementene i A som er slik at hvis a er forskjellig fra b og $a \leq b$, så kommer a foran b i rekkefølgen.

Eksempel 5 La $A = \{2,3,4,6,8,9,10,12\}$ ha ordningen definert i *Eksempel* 1. Da svarer rekkefølgen i A til en toplogisk sortering. Men dette er ikke entydig. Hvis vi isteden tar rekkefølgen $\{3,2,6,4,9,10,12,8\}$, har vi også en topologisk sortering. Alle rekkefølger der 2 kommer foran 4, 6 og 10, 3 kommer foran 6 og 9, 4 kommer foran 8 og 12 og til slutt 6 kommer foran 12, vil gi en toplogisk sortering.

Vi kan bruke en komparator laget for en delvis ordning til å sortere toplogisk. Men vi kan kan ikke bruke en hvilken som helst sorteringsmetode. Vi kan sortere en tabell slik: Finn først et maksimalt element a. Det er mulig siden en tabell inneholder endelig mange elementer. Flytt, ved hjelp av ombytting, a bakerst. Ethvert annet element b slik at $b \leq a$ vil da måtte havne til venstre for a og dermed ligger a rett plassert med hensyn på en topologisk sortering. Finn så et maksimalt element blant resten og flytt det nest bakerst. Da vil også det ligge på rett plass med hensyn på en topologisk sortering. Osv. Men dette er jo nøyaktig samme idé som i *utvalgssortering*. Dermed kan vi bruke en komparatorversjon av utvalgssortering (se *Oppgave* 2a i *Avsnitt* 1.4.9) til å utføre en topologisk sortering.

Flg. eksempel viser hvordan vi kan gjøre en toplogisk sortering av heltall med hensyn på den delvise ordningen som er beskrevet i *Eksempel* 1:

```
Integer[] a = {3,8,2,10,9,12,6,4};

Comparator<Integer> c = (x, y) ->
{
  if (x == y) return 0;                 // x og y er like
  else if (y % x == 0) return -1;       // x går opp i y
  else if (x % y == 0) return 1;        // y går opp i x
  else return 0;                        // ikke sammelignbare
};

Tabell.utvalgssortering(a, c);
System.out.println(Arrays.toString(a));
// Utskrift: [2, 10, 4, 8, 3, 6, 12, 9]
```

Programkode* 1.4.13 *d)

Vi kan også lage en topologisk sortering av ordningen i *Eksempel* 2:

```
Comparator<Tallmengde> c = (A, B) ->
{
  if (A.equals(B)) return 0;            // A og B er like
  else if (B.inklusjon(A)) return -1;   // A delmengde av B
  else if (A.inklusjon(B)) return 1;    // B delmengde av A
  else return 0;                        // ikke sammelignbare
};

int[][] A = { {1,2}, {2,3}, {3}, {1,2,3}, {1}, {1,3}, {2}, {} };

Tallmengde[] a = new Tallmengde[A.length];
for (int i = 0; i < A.length; i++) a[i] = new Tallmengde(A[i]);

Tabell.utvalgssortering(a, c);
System.out.println(Arrays.toString(a));

// Utskrift: [{}, {3}, {1}, {1,3}, {2}, {2,3}, {1,2}, {1,2,3}]
```

Programkode* 1.4.13 *e)

Oppgaver til Avsnitt 1.4.13

1. *Programkode* 1.4.13 *b)* finner et maksimalt element i en gitt tallmengde. Hva blir resulatet hvis 1) {2,9,3,10,4,6,8,12}, 2) {3,2,4,8,9,10,6,12} og 3) {2,3,4,6,12,8,9,10} brukes som mengder. Finn først svaret uten å kjøre programmet og sjekk så at programmet gir samme svar.
2. *Programkode* 1.4.13 *d)* sorterer en tallmengde topologisk ved hjelp av utvalgssortering. Hva blir resultatet hvis du bruker de tre tallmengdene i *Oppgave* 1?
3. La metoden *randPermInteger* i *Programkode* 1.4.3 *d)* lage tilfeldige `Integer`-tabeller med tallene fra 1 til 20. Sorter dem slik som i *Programkode* 1.4.13 *d)*. Sjekk at det blir en topologisk sortering.

1.4.14 Referanser

1. Angelika Langer, *Java Generics FAQ*