C Samleklasser

C.1 Instans- og klassemetoder

I Java er en klasse det minste eller laveste selvstendige programmeringsnivået. I enkelte andre språk (for eksempel C og C++) kan en lage selvstendige eller frittstående metoder (globale metoder). Men i Java må enhver metode høre til i en klasse. Det er imidlertid fullt mulig å lage en klasse som har kun én metode og ikke noe annet.

I *Delkapittel* 1.1 ble det laget flere metoder som «er seg selv nok». Dvs. metoder som gjør sine oppgaver kun ved hjelp den informasjonen som kommer via parameterverdier. Ta f.eks metoden `bytt` som bytter om (eng: swap) innholdet i to tabellelementer. Den kunne vi legge inn som eneste metode i en klasse. La f.eks. klassen hete `Bytt`:

```
public class Bytt
{
  public void bytt(int[] a, int i, int j)  // bytter om tabellelementer
  {
    int temp = a[i]; a[i] = a[j]; a[j] = temp;
  }
}
```

Programkode* C.1 *a)

Legg merke til at metoden `bytt` nå ikke er deklarert som *statisk* (eng: static). Klassen `Bytt` har heller ingen variabler. Hvis vi nå får bruk for å bytte om elementer i en tabell, må det først lages et `Bytt`-objekt før vi kan få tak i metoden:

```
int[] a = {1,2,3,4,5,6,7,8,9,10};        // en heltallstabell
Bytt b = new Bytt();                     // b er et Bytt-objekt
b.bytt(a, 0, 9);                         // bytter om første og siste element
System.out.println(Arrays.toString(a)); // skriver ut

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

Programkode* C.1 *b)

I klassen `Bytt` er nå metoden *bytt* en *instansmetode*. Det betyr at hver instans av klassen `Bytt` får sin egen versjon av metoden. Dette er helt unødvendig siden metoden er helt uavhengig av instansen. Den er kun avhengig av metodens parameterverdier. Vi lar derfor metoden være statisk, dvs. slik:

```
public class Bytt
{
  public static void bytt(int[] a, int i, int j)  // bytter om tabellelementer
  {
    int temp = a[i]; a[i] = a[j]; a[j] = temp;
  }
}  // Bytt
```

Programkode* C.1 *c)

Når metoden *bytt* nå er deklarert som en statisk metode i klassen `Bytt`, betyr det at den er en *klassemetode*. Da hører metoden til klassen som sådan og ikke til de enkelte instansene. *Programkode* C.1 *b)* vil virke som før, men siden metoden nå hører til klassen kan vi (og skal vi) benytte den ved å referere til klassen:

```
int[] a = {1,2,3,4,5,6,7,8,9,10};        // en heltallstabell
Bytt.bytt(a, 0, 9);                      // refererer til klassen Bytt
System.out.println(Arrays.toString(a)); // skriver ut

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

Programkode* C.1 *d)

Hvis klassen `Bytt` skal ha kun denne metoden, gir det ingen mening å lage instanser av den. En klasse får automatisk en standardkonstruktør (eng: default constructor), dvs. en uten parametre. Det er den konstruktøren som ble brukt i *Programkode* C.1 *b)*. Hvis det ikke er aktuelt eller ønskelig å lage instanser av en klasse, er det en vanlig teknikk å «blokkere» standardkonstruktøren. Det gjøres ved å deklarere den som privat:

```
public class Bytt
{
  private Bytt() { }      // en privat standardkonstruktør

  public static void bytt(int[] a, int i, int j)  // bytter om tabellelementer
  {
    int temp = a[i]; a[i] = a[j]; a[j] = temp;
  }
}  // Bytt
```

Programkode* C.1 *e)

Hvis en bruker denne versjonen av klassen `Bytt`, vil ikke lenger *Programkode* C.1 *b)* la seg kompilere. Prøv! Men *Programkode* C.1 *d)* vil selvfølgelig virke.

C.2 Samleklassene `Arrays` og `Collections`

I *Avsnitt* 1.2.2 startet vi oppbyggingen av samleklassen `Tabell`. Den skal inneholde metoder som arbeider med tabeller. Alle metodene må imidlertid være «seg selv nok», dvs. at de gjør jobben sin kun ved hjelp av den informasjonen de får via parameterverdier. Dermed kan de deklareres som statiske metoder (klassemetoder). F.eks. ble metoden *bytt* som vi diskuterte i avsnittet over, lagt inn der. På samme måte som for klassen `Bytt`, ble klassen `Tabell` utstyrt med en privat standardkonstruktør.

Java har de to samleklassene `Arrays` og `Collections`. Vår samleklasse `Tabell` vil ha mange metoder av samme type som dem i `Arrays` og mange andre. `Arrays` inneholder kun statiske metoder og har en privat standardkonstruktør. Her er en kort metodeoversikt for `Arrays`:

- binærsøk i sorterte tabeller (og tabellintervaller)
- kopiering og «utvidelser» av tabeller (og tabellintervaller)
- sammenligning av tabeller
- utfylling av tabeller (og tabellintervaller)
- «hashing» av tabeller
- sortering av tabeller (og tabellintervaller)
- toString-metoder for tabeller
- konvertering fra tabeller til lister og strømmer
- teknikker for parallellisering

Uheldigvis har ikke klassen Arrays metoder som genererer tilfeldige permutasjoner av hele tall eller metoder som permuterer innholdet i tabeller. Den har heller ingen maks- og min-metoder for tabeller. Klassen Collections har imidlertid slike metoder, men formelt kun for lister eller for subklasser av Collection. Men en objekttabell kan «legges inn» i en *ArrayList* ved hjelp av metoden *asList* i klassen Arrays. Legg merke til at Collections er en samleklasse for metoder som arbeider med subklasser til grensesnittet Collection. Denne samleklassen har mange forskjellige (statiske) metoder. Her ser vi på noen få av dem.

Shuffle Vi kan permutere innholdet i en objekttabell, f.eks. av typen *Integer*[], ved hjelp av Arrays.*asList* og metoden *shuffle* i Collections:

```
Integer[] a = {1,2,3,4,5,6,7,8,9,10};      // en Integer-tabell
Collections.shuffle(Arrays.asList(a));     // bruker asList og shuffle
System.out.println(Arrays.toString(a));    // utskrift av tabellen a
```

Programkode* C.2 *a)

Maks Vi kan finne den største verdien i en objekttabell (av typen *Comparable*), f.eks. av typen *Character*[], ved hjelp av Arrays.*asList* og metoden *max* i Collections. Da finner vi ikke posjonen, men selve verdien:

```
Character[] c = {'C','F','A','G','D','H','B','E'};      // en Character-tabell
System.out.println(Collections.max(Arrays.asList(c)));  // bruker asList og max

// Utskrift: H
```

Programkode* C.2 *b)

I *Delkapittel* 1.1 var det å finne posisjonen til den største verdien i en heltallstabell brukt som en gjennomgående problemstilling. Det ble bl.a. bestemt (se *Avsnitt* 1.1.7) at hvis tabellen var tom (lengde lik 0), skulle det kastes en *NoSuchElementException*. Hva gjør metoden *max* i klassen Collections? Kjør flg. programbit og se hva som skjer:

```
Character[] c = {};                                     // en tom tabell
System.out.println(Collections.max(Arrays.asList(c)));  // bruker asList og max
```

Programkode* C.2 *c)

Det er også mulig å finne den største ved hjelp av en Stream. Klassen Arrays har metoder som lager en strøm ved hjelp av en tabell:

```
Character[] c = {'C','F','A','G','D','H','B','E'};      // en Character-tabell
System.out.println(Arrays.stream(c).max(Comparator.naturalOrder()).get());

// Utskrift: H
```

Programkode* C.2 *d)

Swap Vi kan bytte om to verdier i en objekttabell, f.eks. av typen Integer[], ved hjelp av Arrays.*asList* og metoden *swap* i Collections:

```
Integer[] a = {1,2,3,4,5,6,7,8,9,10};      // en Integer-tabell
Collections.swap(Arrays.asList(a),0,9);    // bruker asList og swap
System.out.println(Arrays.toString(a));    // utskrift av tabellen a

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

Programkode* C.2 *e)

Reverse Vi kan snu rekkefølgen på verdiene i en objekttabell, f.eks. av typen `String[]`, ved hjelp av `Arrays.asList` og metoden *reverse* i `Collections`:

```
String[] s = {"Ali","Berit","Carl","Doris","Elin"};  // en String-tabell
Collections.reverse(Arrays.asList(s));               // bruker asList og reverse
System.out.println(Arrays.toString(s));              // utskrift av tabellen s

// Utskrift: [Elin, Doris, Carl, Berit, Ali]
```

***Programkode* C.2)**