A.3 Filbehandling på *byte*-nivå

A.3.1 Sammendrag

En datastrøm (eng: data stream) er en sekvens av verdier eller her en sekvens med byter hvis vi er på *byte*-nivå. Det mest vanlige er at denne sekvensen befinner seg på en ekstern enhet (f.eks. harddisk, CD, minnepinne eller internett) i form av en fil.

Java har klassehierarkiene `InputStream` og `OutputStream` for lesing og skriving på *byte*-nivå. Vi skiller normalt mellom formaterte og uformaterte datastrømmer. Strømen er formatert hvis bytene har en kjent sammensetning eller struktur - hvis ikke kalles den uformatert. De fleste av klassene arbeider uformatert, dvs. bytene leses eller skrives enkeltvis. To av dem, dvs. `DataInputStream` og `DataOutputStream`, er imidlertid laget for formaterte strømmer.

Både `InputStream` og `OutputStream` er abstrakte klasser og kan derfor ikke instansieres.

Klassen `FileInputStream` som er direkte subklasse til `InputStream`, er den grunnleggende klassen for lesing av filer på *byte*-nivå. De mest brukte metodene er:

```
public FileInputStream(String fileName)      // konstruktør
public int read()                            // leser en byte
public int read(byte[], int off, int len)    // leser inn i en tabell
public int available()                       // antall byter som er igjen
public void close()                          // lukker
```

Programkode A.3.1 a)

Konstruktøren `FileInputStream(String fileName)` åpner filen *fileName* og *read*() leser én byte om gangen. Den returneres som de 8 siste bitene i en *int* (verdi fra 0 til 255). Hvis det ikke er flere byter igjen, returneres -1 (End Of File). Metoden `int` *available*() forteller til enhver tid hvor mange byter som er igjen, og `void` *close*() lukker filen.

Alle metodene i `A.3.1 a)` kaster unntak. Konstruktøren kaster en `FileNotFoundException` hvis det ikke finnes noen fil med med oppgitte navn. De andre kaster en `IOException` hvis det oppstår en feil på filen. Slike må enten fanges opp i en `try - catch` eller bli sendt videre siden de er av typen «sjekket» unntak. Se forøvrig *Vedlegg* D om unntakshåndtering.

Det er mest effektivt å bruke en buffer-teknikk. Da hentes mange byter om gangen og disse mellomlagres i et internt buffer. Dermed kan *read*() hente én og én byte fra bufferet. Når det er tomt (alle bytene har blitt lest) fylles det opp på nytt. Osv. `BufferedInputStream` som er en subklasse til `InputStream`, bruker en slik teknikk. De mest brukte metodene er:

```
public BufferedInputStream(InputStream in)  // konstruktør
public int read()                           // leser en byte
public int read(byte[], int off, int len)   // leser inn i en tabell
public int available()                      // antall byter som er igjen
public void close()                         // lukker
```

Programkode A.3.1 b)

Flg. program finner først antallet byter på en filen "inn.txt" ved å bruke *available*() og finner så det samme antallet ved å lese hele filen (dvs. telle opp bytene):

```
import java.io.*;  // eller hver enkelt av de aktuelle klassene

public class Program
{
  public static void main(String... args) throws IOException
  {
    InputStream inn = new BufferedInputStream(new FileInputStream("inn.txt"));
    System.out.println("Antall byter = " + inn.available());

    int antall = 0;
    while (inn.read() != -1) antall++;   // teller opp
    inn.close();

    System.out.println("Antall byter = " + antall);
  }
}
```

Programkode A.3.1 c)

I *Programkode* A.3.1 c) over blir unntakene (`FileNotFoundException` og `IOException`) sendt videre ved at `main` kaster `IOException`. En mer «moderne» måte er:

```
public static void main(String... args) throws IOException
{
  int antall;
  try (InputStream inn =
    new BufferedInputStream(new FileInputStream("inn.txt")))
  {
    System.out.println("Antall byter = " + inn.available());
    antall = 0;
    while (inn.read() != -1) antall++;   // teller opp
  }

  System.out.println("Antall byter = " + antall);
}
```

Programkode A.3.1 d)

Klassen `FileOutputStream` som er direkte subklasse til `OutputStream`, er den grunnleggende klassen for skriving til fil på *byte*-nivå. De mest brukte metodene er:

```
public FileOutputStream(String fileName)  // konstruktør
public void write(int b)                  // skriver en byte
public void close()                       // lukker
```

Programkode A.3.1 e)

Konstruktøren oppretter en «datastrøm» med oppgitt navn. Metoden *write*(`int b`) skriver en byte (den siste byten i *int*-parameteren *b*) og *close*() lukker «strømmen».

Konstruktøren kaster en `FileNotFoundException` hvis filen eller «strømmen» ikke lar seg opprette. Navnet på unntaket er litt misvisende. Det kastes ikke noe unntak hvis det ikke finnes en fil med det oppgitte navnet. I det tilfellet blir en slik fil opprettet. Hvis den allerede finnes, blir den fjernet og en ny med samme navn blir opprettet. En årsak til en eventuell `FileNotFoundException` kan f.eks. være at filnavnet er ulovlig eller det er feil på den enheten

der filen skal opprettes. Men uansett må unntaket enten fanges ved hjelp av en `try - catch` eller bli sendt videre. De to andre metodene kaster `IOException`.

Også ved utskrift ter det gunstig å bruke en buffer-teknikk. Metoden *write*(int b) kan da skrive (eller legge inn) byter i et buffer (en *byte*-tabell), og når det er fullt, «tømmes» det, dvs. hele bufferinnholdet skrives ut til fil ved hjelp av en eneste skriveoperasjon. Osv. `BufferedOutputStream` som er en subklasse til `OutputStream`, bruker en slik teknikk. De mest brukte metodene er:

```
public BufferedOutputStream(OutputStream out)  // konstruktør
public void write(int b)                       // skriver en byte
public void flush()                            // tømmer bufferet
public void close()                            // tømmer og lukker
```

Programkode A.3.1 *f)*

Flg. program kopierer en fil ved at de leste bytene fortløpende skrives til en annen fil:

```
public static void main(String... args) throws IOException
{
  InputStream inn = new BufferedInputStream(new FileInputStream("inn.txt"));
  OutputStream ut = new BufferedOutputStream(new FileOutputStream("ut.txt"));

  int verdi;
  while ((verdi = inn.read()) != -1) ut.write(verdi);  // leser og skriver

  inn.close();  // lukker inn-filen
  ut.close();   // lukker ut-filen
}
```

Programkode A.3.1 *g)*

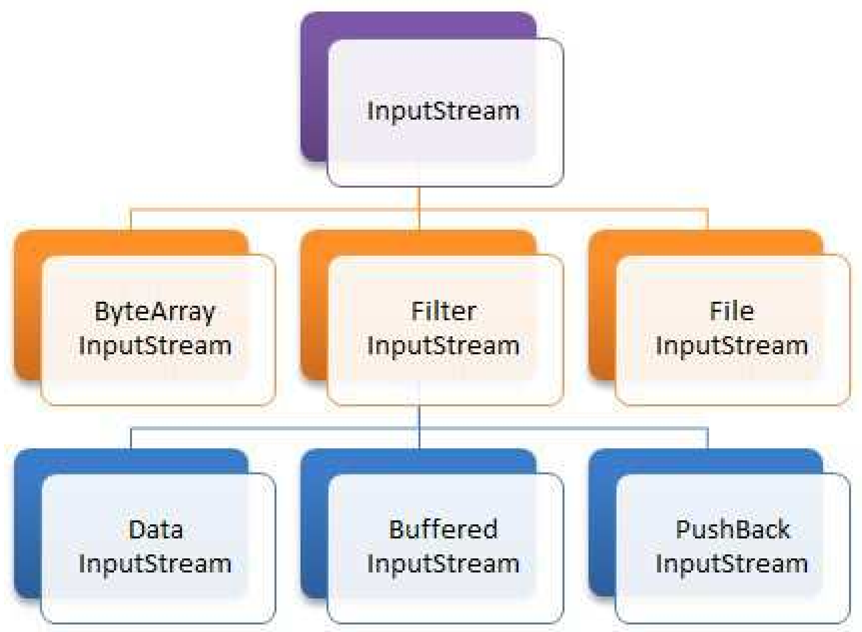
Det er viktig at en instans av `BufferedOutputStream` lukkes ved hjelp av *close*(). Det interne bufferet tømmes automatisk kun når det er fullt, mens *flush*() tømmer bufferet uansett hvor mye det inneholder. I *close*() gjøres det et implisitt kall på *flush*() før filen lukkes. Det er derfor unødvendig for oss å ha et kall på *flush*() før *close*().

Oppgaver til A.3.1

1. I eksemplene over inngår filnavnene inn.txt og ut.txt. Hvis en kjører *Programkode* A.3.1 *c)* uten å ha en fil med navn inn.txt på det gjeldende (eng: current) området, vil en få en `FileNotFoundException`. Sjekk at det stemmer.
2. En kan oppgi hele veien (eng: path) til filen, f.eks. C:/NetBeansAlgDat/AlgDat/inn.txt. Prøv det!
3. Hvis en kjører *Programkode* A.3.1 *g)*, så vil det, hvis det ikke finnes noen fil fra før med navn ut.txt, bli opprettet en slik fil. Hvis det på forhånd finnes en slik fil, vil det bli opprettet en ny fil med samme navn og den gamle forsvinner. Test det ved å kjøre *Programkode* A.3.1 *g)* flere ganger med forskjellige inn-filer. Sjekk også at ut-filen blir en kopi av inn-filen.

A.3.2 InputStream

`InputStream` er en *abstrakt* basisklasse for de klassene som leser data på *byte*-nivå. Den definerer det minimum av metoder som slike klasser bør ha. På tegningen under er en del av klassehierarkiet til `InputStream` satt opp:



Figur A.3.2 a) Noen av subklassene til InputStream

Den abstrakte klassen `InputStream` inneholder flg. offentlige metoder:

```
public abstract class InputStream
{
  public abstract int read() throws IOException
  public abstract int read() throws IOException
  public int read(byte[] b) throws IOException
  public int read(byte[] b, int off, int len) throws IOException
  public byte[] readAllBytes() throws IOException
  public byte[] readNBytes(byte[] b, int off, int len) throws IOException
  public long skip(long n) throws IOException
  public int available() throws IOException
  public void close() throws IOException
  public synchronized void mark(int readlimit)
  public synchronized void reset() throws IOException
  public boolean markSupported()
  public long transferTo(OutputStram out) throws IOException
}
```

Programkode A.3.2 a)

Det er bare metoden `int read()` som er abstrakt. Alle de andre er konkrete, dvs. de har enten fått en «dummy» implementasjon (dvs. metoden gjør ingen ting) eller blit kodet ved hjelp av `int read()`. Det er meningen at alle (eller de fleste av dem) skal overstyres (eng: override), dvs. reimplementeres i eventuelle subklasser.

Hensikten fremgår av navnene. Den første metoden - *read*() - skal lese én *byte* og returnere den som et *int*-tall i intervallet [0,255]. Datatypen *int* består av 4 byter eller 32 biter. Det betyr at den leste byten ligger bakerst i dette heltallet og at de 3 første bytene er 0-byter. Hvis det ikke er flere byter å lese skal den returnere -1, dvs. «End Of File».

Metoden int *read*(byte[] b) skal fylle tabellen *b* med byter, mens int *read*(byte[] b, int off, int len) skal fylle den med (fra og med indeks *off*) *len* antall byter. Begge skal returnere det antallet byter som ble lest inn eller -1 hvis det ikke er flere byter å lese.

I Java 1.9 har klassen fått tre nye lesemetoder. Det er *readAllBytes*(), *readNBytes*() og *transferTo*(). Metoden *readAllBytes*() leser resten av strømmen og returnerer det som en *byte*-tabell. Hvis dette er den første lesemetoden som kalles, vil den returenere hele strømmen. Metoden *readNBytes*() oppfører seg omtrent som *read*-metoden med samme paeameterliste. Den siste, dvs. *transferTo*(), flytter innholdet av en strøm over i en annen strøm (en OutputStream).

Metodene *skip*(), *available*() og *close*() er vel selvforklarende.

De tre metodene *mark*(), *reset*() og *markSupported*() er satt opp for å kunne gi oss den muligheten å hoppe tilbake i datastrømmen, dvs. lese noe om igjen.

Legg merke til de tre direkte subklassene til InputStream. Subklassen FilterInputStream. har kun denne konstruktøren:

```
protected FilterInputStream(InputStream in)
{
  this.in = in;
}
```

Klassen er laget for å kunne arves. En InputStream er parameter i konstruktøren og den lagres i instansvariabelen *in*. Hensikten er kunne lage subklasser som tar utgangspunkt i en InputStream og lager ytterligere funksjonalitet ved hjelp av dens metoder. En slik klasse kalles en omslagsklasse (eng: wrapper class). Et eksempel er BufferedInputStream som vi skal se nærmere på i *Avsnitt* A.3.4.

Subklassen FileInputStream. er den grunnleggende klassen for lesing av *filer* på *byte*-nivå. Den diskuteres i *Avsnitt* A.3.3.

Subklassen ByteArrayInputStream bruker en *byte*-tabell som kilde for lesingen. Den diskuteres i *Avsnitt* A.3.5.

A.3.3 FileInputStream

Klassen FileInputStream, som er en direkte subklasse til InputStream (se *Figur* A.3.2 a), er den grunnleggende klassen for lesing av *filer* på *byte*-nivå. Den tas opp i

Klassen har tre konstruktører:

```
public FileInputStream(String name) throws FileNotFoundException
public FileInputStream(File file) throws FileNotFoundException
public FileInputStream(FileDescriptor fdObj)
```

Programkode* A.3.3 *a)

Den første er mest brukt. Hvis f.eks. "fil.txt" ligger på gjeldende (current) område, holder det med navnet. Hvis ikke, må vi oppgi hele veien - f.eks. "c:/abc/fil.txt":

```
InputStream inn = new FileInputStream("c:/abc/fil.txt");
```

De to andre konstruktørene brukes mer sjelden. Men hvis vi allerede har en instans av klassen File, vil den (så sant den representerer en fil og ikke en mappe (directory)) kunne gi oss en FileInputStream (se også *Vedlegg* A.6):

```
File fil = new File("c:/abc/fil.txt");
InputStream inn = new FileInputStream(fil);
```

De to første konstruktørene kaster FileNotFoundException hvis det ikke finnes noen fil med det navnet som er oppgitt eller hvis filen ikke lar seg åpne. Dette unntaket må enten fanges opp i en try - catch eller bli sendt videre. Det samme gjelder hvis vi vil bruke en av metodene (se *Programkode* A.3.2 a) som kaster en IOException.

FileInputStream er som sagt en subklasse av InputStream og har derfor alle metodene i A.3.2 a). I kildekoden til FileInputStream er en del av dem kodet. Men ser en nøyere etter, ser en at de er kodet ved hjelp av private *native*-metoder. Ta f.eks. *read*():

```
public int read() throws IOException { return read0(); }

private native int read0() throws IOException;
```

Det at en metode er *native* betyr at den ikke er implementert i Java. Når et Java-program kompileres blir det laget *bytekode* (eng: byte code) og denne koden må tolkes av en Java-maskin (JVM) for å bli kjørbar. En JVM lages for en bestemt plattform, f.eks. for Windows. En må kjenne operativsystemet for kunne kode metoder for filbehandling. Slik kode inngår derfor i JVM-en og vil nok være kodet så optimalt som mulig (f.eks. i C) for denne plattformen. Vi må bare ta det som gitt at alle *native*-metodene i FileInputStream virker som beskrevet.

Det er fullt mulig å lese en hel fil ved å lese én og én byte ved hjelp av metoden *read*(). I flg. eksempel bestemmes antall byter på filen "fil.txt" ved at bytene telles opp:

```
InputStream inn = new FileInputStream("c:/abc/fil.txt");
int antall = 0;
while (inn.read() != -1) antall++;                // leser en og en
System.out.println("Antall byter = " + antall);  // antall byter
inn.close();
```

Programkode* A.3.3 *b)

Men dette er langt fra optimalt. Det er mye bedre å bruke en metode som fyller en hel tabell med byter om gangen (se A.3.2 a). I flg. eksempel blir også alle bytene på filen lest (husk at int *read*(byte[] b) returnerer det antallet byter som blir lagt inn i b):

```
InputStream inn = new FileInputStream("c:/abc/fil.txt");
byte[] b = new byte[4096];   // en hjelpetabell på 4 kB
int antall = 0, n;
while ((n = inn.read(b)) != -1) antall += n;
System.out.println("Antall byter = " + antall);
inn.close();
```

Programkode A.3.3 c)

Programkode A.3.3 c) er langt mer effektiv enn den i A.3.3 b). Hvor mye mer er avhengig av plattform og av type enhet det leses fra, men den er nok av størrelsesorden mange hundre ganger raskere. Test selv! Bruk en ganske stor fil (gjerne flere megebyte). Se *Oppgave* 1.

Det er ikke helt rettferdig å sammenligne A.3.3 b) med A.3.3 c) siden den siste ikke ser på hver enkelt byte. I flg. eksempel finner vi antall forekomster av tegnet 'A' på to måter:

```
// 1. versjon - bruker bare read()
int antall = 0, k;
while ((k = inn.read()) != -1) if (k == 'A') antall++;

// 2. versjon - bruker read(byte[] b)
byte[] b = new byte[4096];  // hjelpetabell på 4 kb
int antall = 0, n;
while ((n = inn.read(b)) != -1)
{
  for (int i = 0; i < n; i++) if (b[i] == 'A') antall++;
}
```

Programkode A.3.3 d)

Også her vil 2. versjon være svært mye raskere enn 1. Test selv! Se *Oppgave* 1. Typen enhet er avgjørende. Med en «roterende» enhet (vanlig harddisk og CD), er det svært stor forskjell. En bestemt byte leses når «lesehodet» passerer byten. Men tar omtrent like mye tid å lese alle bytene som «lesehodet» passerer under samme rotasjon. Hvor mange som kan leses under én rotasjon er plattformavhengig. FileInputStream har en *native*-metode for dette:

```
private native int readBytes(byte b[], int off, int len) throws IOException;
```

I FileInputStream er **int** *read*(byte[] b) er kodet ved hjelp av den. I 2. versjon i A.3.3 d) brukes en tabell på 4kb. Om den fylles med kun én leseoperasjon eller om det trengs flere, er nok plattformavhengig. Men den fylles helt sikkert med få leseoperasjoner. Når vi så trenger de enkelte bytene, er det bare å hente dem ut fra tabellen. Obs: Hvis vi bruker en annen type enhet, f.eks. flashminne, så er ikke forskjellen mellom så stor. Da er 2. versjon kanskje bare 10 - 20 ganger så rask? Det kommer av at et flashminne er helt annerledes organisert. Se også neste avsnitt om BufferedInputStream.

Oppgaver til A.3.3

1. Vi finner tidsforbruket ved lese av systemklokken (System.currentTimeMillis) på forhånd og så etterpå. Forskjellen gir tidsforbruket. Men her må en normalt bruke ganske store filer for få målbare resutater. Bruk både roterende minne og flashminne.

A.3.4 BufferedInputStream

BufferedInputStream er en subklasse av FilterInputStream (se *Figur* A.3.2 *a)*) og er en omslagsklasse (wrapper class) for en InputStream. Den bruker en intern *byte*-tabell (slik som i *Programkode* A.3.3 *c*). Den har to konstruktører:

```
public BufferedInputStream(InputStream in, int size)
{
  super(in);  // konstruktøren til FilterInputStream
  if (size <= 0) throw new IllegalArgumentException("Buffer size <= 0");
  buf = new byte[size];
}

public BufferedInputStream(InputStream in)
{
  this(in, DEFAULT_BUFFER_SIZE);
}
```

Programkode* A.3.4 *a)

En InputStream-instans *in* er parameter i begge konstruktørene. En *byte*-tabell med navn ***buf*** fungerer som internt buffer. Første konstruktør lar oss bestemme størrelsen (parameter *size*) selv, mens den andre bruker en standardstørrelse (***DEFAULT_BUFFER_SIZE***). Hvor stor bufferstørrelse bør vi velge? Hvis den underliggende datastrømmen er en FileInputStream bør den være ganske stor. Det er plattformavhengig hva som er optimal størrelse, men i Java 1.8 er ***DEFAULT_BUFFER_SIZE*** på 8 kb.

Metoden *read*() som leser én byte om gangen, virker slik: Hvis ***buf*** er tom, blir den først fylt opp ved at byter hentes fra den underliggende datastrømmen. I en FileInputStream er det da den private *native*-metoden *readBytes*() som gjør jobben. Deretter returnerer *read*() den første byten i ***buf***. Ved senere kall på *read*() hentes byter fortløpende fra ***buf*** inntil alle er hentet ut. Deretter vil vil den bli fylt opp på nytt. Osv.

Teknikken med å fylle et helt buffer om gangen i steden for å lese én og én byte vil normalt føre til en dramatisk bedre effektivitet. Vi bør derfor alltid bruke en BufferedInputStream (eventuelt en BufferedReader på *char*-nivå - se *Vedlegg* A.4) ved fillesing.

I *Programkode* A.3.3 d) (2. versjon) fant vi antall A-er ved å bruke vår egen *byte*-tabell. Men i slike tilfeller bruker vi normalt en BufferedInputStream (se nedenfor) selv om det ikke er fullt så effektivt. Hvert *read*-kall koster litt, mens vi i *Programkode* A.3.3 d) henter bytene direkte fra tabellen:

```
InputStream inn =
  new BufferedInputStream(new FileInputStream("e:/abc/fil.txt"));

int antall = 0, k;
while ((k = inn.read()) != -1) if (k == 'A') antall++;

System.out.println("Antall byter = " + antall);
inn.close();
```

Programkode* A.3.4 *b)

BufferedInputStream har alle metodene i *Programkode* A.3.2 a), men ingen flere (offenlige metoder) enn det. Alle har kode. Også metoden *reset*() som gjør at vi kan «hoppe tilbake», dvs. lese noe om igjen. Metoden *markSupported*() returnerer derfor *true*.

Et kall på metoden *mark*(int readlimit) setter et *merke* i datastrømmen (egentlig en indeks i den interne *byte*-tabellen). Et senere kall på *reset*() gjør at lesingen «hopper tilbake» til det merket (eller indeksen). Med andre ord kan vi da lese om igjen det vi leste sist. Men vi har imidlertid en grense *readLimit*. Har vi lest flere byter etter der merket ble satt enn det grensen sier, vil ikke *reset*() lenger virke. I flg. eksempel tenker vi oss at filen "fil.txt" inneholder bokstavene fra A til Z:

```
BufferedInputStream inn =             // bruker 4 som bufferstørrelse
  new BufferedInputStream(new FileInputStream("e:/abc/fil.txt"), 4);
int grense = 10;
inn.mark(grense);                      // merket settes i posisjon 0

for (int i = 0; i < grense; i++) System.out.print((char)inn.read());
System.out.println();  // ny linje

inn.reset();  // lesingen starter om igjen
for (int i = 0; i < grense; i++) System.out.print((char)inn.read());
inn.close();
// Utskrift:
// ABCDEFGHIJ
// ABCDEFGHIJ
```

***Programkode* A.3.4 c)**

I koden over kalles *mark*() før det er gjort noen leseoperasjoner. Så kalles *read*() 10 ganger (*grense* = 10). Deretter kommer *reset*(). Det fører til at lesingen starter fra begynnelsen igjen. Men hvis vi i den første for-løkken bruker *grense* + 1 istedenfor *grense* (da blir det 11 leseoparsjoner), vil *reset*() kaste unntaket: IOException: Resetting to invalid mark. Vi har med andre ord gjort flere leseoperasjoner enn *grense* sier. Se *Oppgave* 2.

Legg også merke til at bufferstørrelsen settes til 4 i Programkode A.3.4 c) over. Men det er lett å forstå at for å kunne hoppe f.eks. 10 enheter tilbake, må alle de aktuelle bytene ligge i bufferet (*byte*-tabellen). Det som derfor skjer hvis den opprinnelige bufferstørrelsen er mindre enn *grense*, er at den vil bli utvidet slik at den garantert har minst så mange byter som *grense* sier. Det betyr at hvis vi i et kall på *mark*() bruker en *grense* som er svært stor, vil også den interne *byte*-tabellen bli tilsvarende stor.

Metoden long skip(long n) (se A.3.2 a) hopper over («skipper») et bestemt antall byter. Det kunne vi ha fått til ved å kalle *read*() samme antall ganger. Men *skip*() gjør dette mer effektivt. Returverdien er antallet byter som ble «skippet».

Oppgaver til A.3.4

1. Lag et program der *Programkode* A.3.4 b) inngår. Sammenlig tidsforbruket med den i 2. versjon av *Programkode* A.3.3 d). Bruk en stor fil - gjerne flere megabyte.
2. Lag en fil som inneholder bokstavene fra A til Z. Bruk den i *Programkode* A.3.4 c). Bruk så *grense* + 1 istedenfor *grense* i den første for-løkken. Hva skjer?
3. Ta utgangspunkt i *Programkode* A.3.4 c), men gjør det slik at først skrives de tyve første bokstavene (A – T) ut, så (på ny linje) de fem første ((A – E) ut på nytt og til slutt (på ny linje) resten av bokstavene (fra U og videre). Bruk *skip*()!
4. Lag kode som skriver ut den siste byten på filen. Hvis filen inneholder bokstavene fra A til Z, blir det Z.

A.3.5 ByteArrayInputStream

ByteArrayInputStream er en subklasse av InputStream - se *Figur* A.3.2 a). En *byte*-tabell er ekstern «datastrøm». ByteArrayInputStream har to konstruktører:

```
public ByteArrayInputStream(byte[] buf)
public ByteArrayInputStream(byte[] buf, int off, int len)
```

Den første konstruktøren ser på hele tabellen *buf* som en «datastrøm», mens den andre tar den delen av *buf* som starter i indeks *off* og går *len* antall byter videre.

En *byte* er en heltallstype som tillater verdier fra -128 til 127. Vi kan sette opp en *byte*-tabell direkte i koden vår og så lese fra den:

```
byte[] buf = {7, -1, 10, 31, -10};
InputStream inn = new ByteArrayInputStream(buf);
int k; while ((k = inn.read()) != -1) System.out.print(k + " ");
// Utskrift: 7 255 10 31 246
```

***Programkode* A.3.5 a)**

Legg merke til at utskriften ikke gir de samme tallene som i tabellen *buf*. Det kommer av at *read*() returnerer byten som de åtte siste bitene i et *int*-tall der de øvrige bitene er 0-biter. F.eks. er tallet -1 som *byte*-verdi representert ved 11111111 på bitnivå. Når de åtte 1-bitene legges bakerst, vil dette bli 255 som *int*-verdi. Men vær oppmerksom på at det ikke er slik når en *byte* konverteres til en *int*. Se flg. eksempel:

```
byte b = -100;
System.out.println(b + " " + (int)b);  // Utskrift: -100 -100
```

***Programkode* A.3.5 b)**

Bitkoden for *byte*-verdien -100 er 10011100 (1 som første bit). I konverteringen legges det da på 24 1-biter foran. Hvis tallet ikke er negativt (dvs. 0 som første bit), legges det på 24 0-biter foran. Det er imidlertid mulig å konvertere en *byte*-verdi til en *int* slik at det alltid blir 24 0-biter foran. Da kalles det en konvertering uten fortegn (unsigned):

```
byte b = -100;
int n = Byte.toUnsignedInt(b);
System.out.println(b + " " + n);  // Utskrift: -100 156
```

***Programkode* A.3.5 c)**

Vi kan bruke en tegnstreng (String) som kilde ved hjelp av metoden *getBytes*():

```
InputStream inn = new ByteArrayInputStream("Dette er en tegnstreng!".getBytes());
int k; while ((k = inn.read()) != -1) System.out.print((char)k);
inn.close();  // Utskrift: Dette er en tegnstreng!
```

***Programkode* A.3.5 d)**

Oppgaver til A.3.5

1. La {-1, -2, -3, -4, -5} være parameter i *Programkode* A.3.5 a). Sjekk utskriften. Konverter så til en *byte* (dvs. (*byte*)k) i utskriftssetningen. Hva blir utskriften nå?
2. Gjør som i Oppgave 2, men slik at bare de 3 siste verdiene i tabellen brukes. Dvs. bruk den andre konstruktøren for ByteArrayInputStream.

A.3.6 PushbackInputStream

PushbackInputStream er en subklasse av FilterInputStream (se *Figur* A.3.2 a) og er en omslagsklasse (wrapper) for InputStream. Denne klassen gir muligheten å kunne legge én eller flere byter inn eller tilbake (push back) en eller flere byter som tidligere har blitt lest, i en datstrøm. PushbackInputStream har to konstruktører:

```
public PushbackInputStream(InputStream in);
public PushbackInputStream(InputStream in, int size);
```

Klassen har en intern hjelpetabell og det er i den som en eventuell tilbakelegging skjer. I den første konstruktøren får hjelpetabellen dimensjon 1. Det betyr at for hver *read*() kan kun én byte (vha. *unread*) legges tilbake. I den andre får hjelpetabellen dimensjon *size*.

PushbackInputStream har flg. metoder (i tillegg til de fra A.3.2 a):

```
public void unread(int b) throws IOException
public void unread(byte[] b) throws IOException
public void unread(byte[] b, int off, int len) throws IOException
```

Flg. eksempel vil vise hvordan en *unread*-metode brukes. Vi lager en ByteArrayInputStream ved hjelp av en tegnstreng (slik som i *Programkode* A.3.5 *d)* og bruker den som underliggende datastrøm for en PushbackInputStream:

```
InputStream is = new ByteArrayInputStream("Dette er en tegnstreng!".getBytes());
PushbackInputStream inn = new PushbackInputStream(is);

int b = inn.read();           // første byte - dvs. 'D' eller 68
System.out.println((char)b);  // gir D som utskrift
inn.unread(b);  // legger den første byten tilbake

while ((b = inn.read()) != -1) System.out.print((char)b);
// Utskrift: Dette er en tegnstreng!
inn.close();
```

Programkode* A.3.6 *a)

I *Programkode* A.3.6 *a)* vil det ikke være lov å kalle *unread*() to ganger på rad. Men hvis det i mellomtiden har vært en *read*() så kan *unread*() kalles igjen. Hvis en har behov for å legge inn flere enn én byte om gangen, må den andre konstruktøren brukes. Se flg. eksempel:

```
InputStream is = new ByteArrayInputStream("Dette er en tegnstreng!".getBytes());
PushbackInputStream inn = new PushbackInputStream(is, 20);  // 20 som dimensjon

byte[] buf = new byte[10];  // hjelpetabell
inn.read(buf,0,8);          // leser "Dette er"

inn.unread(" ikke".getBytes());   // legger inn " ikke"
inn.unread(buf,0,8);              // legger tilbake "Dette er"

for (int b = inn.read(); b != -1; b = inn.read())
{
  System.out.print((char)b);
}
// Utskrift: Dette er ikke en tegnstreng!
inn.close();
```

Programkode* A.3.6 *b)

Oppgaver til A.3.6

1. I *Programkode* A.3.6 *b)* får den interne tabellen i PushbackInputStream dimensjon 20. Prøv andre (og mindre) dimensjoner inntil du finner den minste som trengs for at koden ikke skal kaste et unntak. Forklar, etter at du har funnet det tallet, hvorfor det må være nettopp det tallet som er minimumsdimensjon.

A.3.7 DataInputStream

DataInputStream er en subklasse av FilterInputStream og implementerer grensesnittet DataInput. Den brukes til formatert innlesing. Det betyr at vi i tillegg til å lese *byte*-verdier kan lese *int*-verdier, *double*-verdier, osv. I flg. eksempel brukes en *byte*-tabell som kilde:

```
byte[] buf = {1, 2, 3, 4, 5, 6, 7, 8};  // 8 byter = 64 biter
DataInputStream inn = new DataInputStream(new ByteArrayInputStream(buf));

int i = inn.readInt();            // de første 32 bitene
short s = inn.readShort();        // de neste 16 bitene
byte b = inn.readByte();          // de neste 8 bitene
boolean l = inn.readBoolean();    // de neste 8 bitene

System.out.println(i + " " + s + " " + b + " " + l);
// Utskrift: 16909060 1286 7 true
```

Programkode* A.3.7 *a)

Her kunne vi på forhånd ha funnet ut hva utskriften ville bli. F.eks. er det første heltallet gitt ved $1·256^3 + 2·256^2 + 3·256 + 4 = 16909060$. Datatypen *boolean* bruker 8 biter og er *false* kun hvis alle bitene er 0. De åtte bitene vi har her er 00001000 (tallet 8) og gir dermed *true*. Se også *Oppgave* 1.

DataInputStream og DataOtputStream er de «motsatte» av hverandre. Vi kan bygge opp en formatert fil ved hjelp av metoder fra den siste og så lese den ved hjelp av de «motsatte». Vi ser nøyere på DataOtputStream i *Avsnitt* A.3.12.

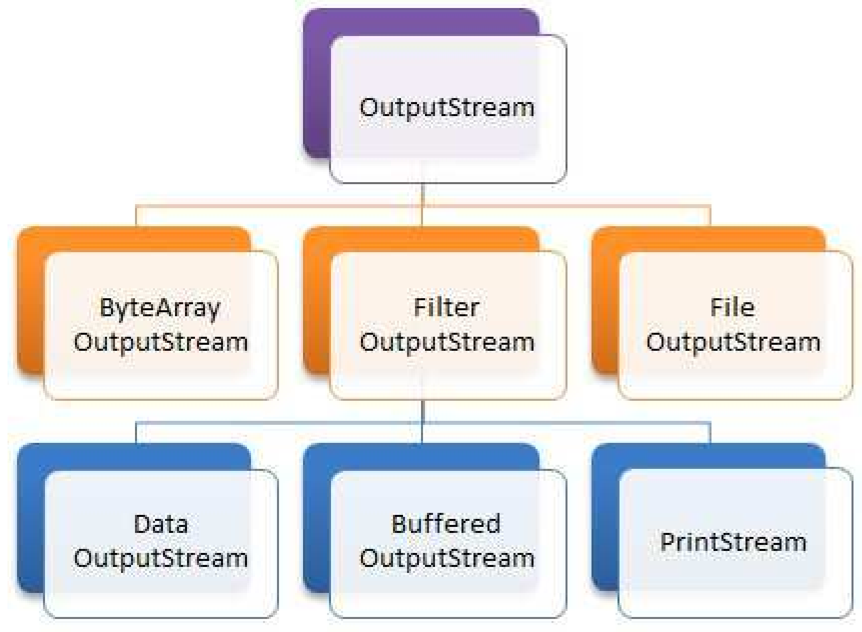
I den tidligste versjonen av Java ble denne klassen også brukt i forbindelse med lesing av tekstfiler, dvs. filer der hver linje avsluttes med linjeskift (LF og CR i Windows). Metoden heter String *readLine*(). Men den er nå utdatert (deprecated). Lesing av tekstfiler bør isteden skje ved hjelp av metoden String *readLine*() i klassen BufferedReader. Se *Avsnitt* A.4.1.

Oppgaver til A.3.7

1. Bytt ut tallet bakerst i tabellen i *Programkode* A.3.7 *a)* med først -8 og så 0. Hva blir utskriften i de to tilfellene. Kunne du på forhånd ha regnet ut hva *short*-verdien vil bli?
2. Bruk tallene fra -1 til -8 istedenfor tallene fra 1 til 8 i *Programkode* A.3.7 *a)*. Klarer du å regne ut hva utskriften vil bli uten å kjøre programmet?
3. Lag en metode som skriver ut innholdet av en todimensjonal *int*-tabell til fil. Tabellen og filnavnet skal være argumenter. Sørg for at tabelldimensjonen skrives ut først. Lag så en metode som leser filen (filnavnet som argument) og returnerer tabellen. Lag til slutt et main-program som tester dette. Bruk DataOtputStream og DataInputStream.

A.3.8 OutputStream

OutputStream er en **abstrakt** basisklasse for de klassene som skriver data på *byte*-nivå. Den definerer det minimum av metoder som slike klasser bør ha. På tegningen under er en del av klassehierarkiet til OutputStream satt opp:



Figur A.3.8 Noen av subklassene til OutputStream

Den abstrakte klassen OutputStream inneholder flg. offentlige metoder:

```
public abstract class OutputStream
{
  public abstract void write(int b) throws IOException;
  public void write(byte b[]) throws IOException;
  public void write(byte b[], int off, int len) throws IOException;
  public void flush() throws IOException;
  public void close() throws IOException;
}
```

Programkode A.3.8 a)

Det er bare metoden void *write*(int b) som er abstrakt. Alle de andre er konkrete, dvs. de er implementert på en eller annen måte. De to siste *write*-metodene er implementert ved hjelp av den første, og metodene *flush*() og *close*() har tom kode. Det er meningen at alle (eller de fleste av dem) skal overstyres (eng: override), dvs. reimplementeres i eventuelle subklasser.

Hva som er hensikten med metodene fremgår av metodenavnene. Den første metoden - void *write*(int b) - skal skrive én *byte*, dvs. den siste byten i *int*-parameteren *b*. Metoden void *write*(byte b[]) skal skrive alle bytene i tabellen *b*, mens void *write*(byte b[], int off, int len) skal skrive *len* antall byter fra *b* (fra og med indeks *off*) . Metoden void *flush*() tømmer et eventuelt internt buffer, og void *close*() lukker filen.

Legg merke til at OutputStream har FilterOutputStream som subklasse (se *Figur* A.3.8) Det er normalt aldri aktuelt å lage instanser av den. Hensikten med klassen er at konstruktører i eventuelle subklasser skal kunne ta en instans av OutputStream som parameterverdi.

A.3.9 FileOutputStream

FileOutputStream som er en direkte subklasse av OutputStream (se *Figur* A.3.8), er den grunnleggende klassen for skriving til filer på *byte*-nivå.

Klassen har fem konstruktører:

```
public FileOutputStream(String name) throws FileNotFoundException;
public FileOutputStream(String name, boolean append) throws FileNotFoundException;
public FileOutputStream(File file) throws FileNotFoundException;
public FileOutputStream(File file, boolean append) throws FileNotFoundException;
public FileOutputStream(FileDescriptor fdObj);
```

Programkode* A.3.9 *a)

Det er den første konstruktøren som er mest brukt. Anta at vi ønsker å skrive til en fil med navn "utfil.xyz". Da brukes konstruktøren slik:

```
OutputStream ut = new FileOutputStream("utfil.xyz");
```

Filen "utfil.xyz" blir opprettet på gjeldende område (eng: current directory). Hvis vi vil ha den på et bestemt område må vi oppgi veien (eng: path) dit, f.eks. "c:/abc/utfil.xyz".

Den andre konstruktøren brukes hvis en vil skrive videre på en fil (eng: append), dvs. skrive etter det som allerede måtte ligge der fra før:

```
OutputStream ut = new FileOutputStream("utfil.xyz", true);
```

Det er mer uvanlig å bruke en av de tre siste konstruktørene. Men hvis vi allerede har en instans av klassen File vil den (så sant den representerer en fil og ikke et område) kunne gi oss en FileOutputStream (se også Delkapittel A.6 om klassen File):

```
File fil = new File("c:/abc/utfil.xyz");
OutputStream ut = new FileOutputStream(fil);
```

Konstruktørene kan kaste en FileNotFoundException og den må enten fanges ved hjelp av en try - catch eller bli sendt videre. Dette skjer f.eks. hvis filnavnet er ulovlig, ikke gir mening eller det er feil på den enheten der filen skal opprettes.

FileOutputStream er en direkte subklasse av OutputStream og har derfor alle metodene i *Programkode* A.3.8 *a)*. Men hvis vi ser på kildekoden til FileOutputStream vil vi se at flere av dem er kodet ved bruk av hjelpemetoder som er «native». Se flg. eksempel:

```
private native void write(int b, boolean append) throws IOException;

public void write(int b) throws IOException
{
  write(b, append);  // bruker metoden over - append er en konstant i klassen
}
```

Programkode* A.3.9 *b)

En *native* metode er ikke implementert som en del av Java. Når et Java-program kompileres blir det laget bytekode (eng: byte code) og denne koden må tolkes av en Java-maskin (JVM) for å bli kjørbar. En JVM lages for en bestemt plattform, f.eks. for Windows. En må kjenne operativsystemet for kunne kode metoder for filbehandling. Slik kode inngår derfor i JVM-en og vil nok være kodet så optimalt som mulig (f.eks. i C) for denne plattformen. Vi må bare ta det som gitt at alle native-metodene i FileOutputStream virker som beskrevet.

Vi kan skrive én og én byte om gangen ved hjelp av *write*(int b):

```
byte[] tabell = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};      // en samling verdier
OutputStream ut = new FileOutputStream("utfil.xyz");  // åpner en fil
for (byte b : tabell) ut.write(b);                    // skriver en og en verdi
ut.close();                                           // lukker filen
```
Programkode A.3.9 c)

Men det er langt mer effektivt å skrive innholdet av en tabell på direkten. Det er tilrettelagt for det ved at de andre *write*-metodene er kodet ved hjelp av flg. private *native*-metode:

```
private native void writeBytes(byte b[], int off, int len, boolean append)
```

Tabellen i *Programkode* A.3.9 *c)* er alt for liten til at vi vil merke en effektivitetsforbedring, men uansett burde det vært kodet slik:

```
byte[] tabell = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};      // en samling verdier
OutputStream ut = new FileOutputStream("utfil.xyz");  // åpner en fil
ut.write(tabell);                        // skriver hele tabellen
ut.close();                              // lukker filen
```
Programkode A.3.9 d)

Hvis vi ikke allerede hadde hatt en tabell, burde vi ha opprettet en og lagret verdier der før vi skrev ut. Men dette slipper vi å tenke på hvis vi isteden bruker en BufferedOutputStream «rundt» en FileOutputStream. Dette ser vi mer på i Avsnitt A.3.10.

Oppgaver til A.3.9

1. FileOutputStream har fem metoder som er *native*. Hvilke det er finner du ved å gå inn i kildekoden til klassen.
2. I *Programkode* A.3.9 *c)* skrives innholdet av en tabell til fil ved at én og én byte skrives ut, mens det i *Programkode* A.3.9 *c)* gjøres ved at hele tabellen skrives ut i et jafs. Lag kode som måler hvor lang tid de to versjonene bruker. Da må du nok bruke en stor tabell, f.eks. størrelse 100000 eller kanskje 1 million. Det holder at tabellen dimensjoneres. Den trenger ikke ha noe annet en standardinnholdet, dvs. 0-verdier.

A.3.10 BufferedOutputStream

BufferedOutputStream er en subklasse av FilterOutputStream og blir dermed «barnebarn» til OutputStream (se Figur A.3.8). Den har to konstruktører:

```
BufferedOutputStream(OutputStream out)
BufferedOutputStream(OutputStream out, int size)
```

OutputStream-instansen *out* som går inn som parameterverdi i begge konstruktørene, blir internt brukt som parameterverdi i et kall på basisklassens konstruktør (*out* kalles klassens *underliggende datastrøm*).

Klassen bruker en *byte*-tabell med navn *buf* som internt buffer. Den første konstruktøren bruker en standardstørrelse for *buf* (i Java 1.8 er den på 8 kb), mens den andre lar oss bestemme (parameter *size*). Det er plattformavhengig hva som er optimal størrelse.

BufferedOutputStream har siden den er subklasse av OutputStream, de tre *write*-metodene i *Programkode* A.3.8 *a)*. Metoden *write*(int b) som skriver en byte, er kodet slik:

```
public synchronized void write(int b) throws IOException
{
  if (count >= buf.length) flushBuffer();
  buf[count++] = (byte)b;
}
```

Programkode* A.3.10 *a)

Instansvariabelen *count* holder rede på hvor mange verdier som er lagt inn i tabellen *buf*. Hvis den er full (dvs. count >= buf.length), blir den «tømt». Hjelpemetoden *flushBuffer*() kaller en *write*-metode (en som skriver en hel tabell) på den underliggende datastrømmen. Hvis tabellen ikke er full, legges verdien *b* på første ledige plass i *buf*. Dette betyr at det ikke er ineffektivt å kalle denne metoden mange ganger, men hvis vi allerede har verdiene i en tabell, bruker vi selvfølgelig en *write*-metode der tabellen inngår som argument.

Det kan hende at det interne bufferet *buf* ikke er fullt når vi skal avslutte. Derfor må vi alltid «tømme» det før vi avslutter. Metoden *flush*() (se *Programkode* A.3.8 *a*) gjør den jobben. Men heldigvis er det ikke nødvendig å kalle *flush*() hvis vi avslutter ved hjelp av *close*() siden det der gjøres et implisitt kall på *flush*() før det hele lukkes.

Flg. kode kopierer en fil ved å lese én byte om gangen fra den ene og så skrive til den andre:

```
InputStream inn = new BufferedInputStream(new FileInputStream("innfil.xyz"));
OutputStream ut = new BufferedOutputStream(new FileOutputStream("utfil.xyz"));

for (int n = inn.read(); n != -1; n = inn.read()) ut.write(n);

inn.close();  ut.close();  // obs: det er viktig å lukke filene
```

Programkode* A.3.10 *b)

Oppgaver til A.3.10

1. I *Programkode* A.3.10 *b)* hadde det vært bedre å lese inn en hel tabell om gangen og så skrive den ut. Gjør om koden slik at det skjer.

A.3.11 ByteArrayOutputStream

ByteArrayOutputStream er en direkte subklasse til OutputStream (se Figur A.3.8) og kan ses på som den «omvendte» klassen til ByteArrayInputStream (se Avsnitt A.3.5). Den bruker en *byte*-tabell som «underliggende datastrøm».

ByteArrayOutputStream har, siden den er en subklasse av OutputStream, alle metodene som er satt opp i *Programkode* A.3.8 *a)*. I tillegg har den flg. konstruktører og metoder (Obs: Alle metodene er ***synchronized***):

```
public ByteArrayOutputStream();                // konstruktør
public ByteArrayOutputStream(int size);        // konstruktør
public void reset();
public int size();
public byte[] toByteArray();
public String toString(int hibyte);            // utgått (deprecated)
public String toString();
public String toString(String charsetName) throws UnsupportedEncodingException;
public void writeTo(OutputStream out) throws IOException;
```

Programkode* A.3.11 *a)

write-metodene skriver til en intern *byte*-tabell med 32 som standard dimensjon. Verdiene legges fortløpende inn i tabellen og hvis det ikke er plass, blir den «utvidet». Metoden *size*() forteller hvor mange verdier som er lagt inn og *toByteArray*() gir en kopi av innholdet. Metoden String *toString*() gir innholdet som en tegnstreng og den siste metoden, dvs. *writeTo*(OutputStream out), skriver det hele til en utstrøm. Metoden *reset*() nullstiller.

Flg. kodebit viser hvordan noen av metodene kan brukes:

```
ByteArrayOutputStream ut = new ByteArrayOutputStream();

byte[] verdier = "ABCDabcd".getBytes();         // fra tegnstreng til byter
ut.write(verdier);                              // hele tabellen skrives ut

String innhold = ut.toString();                 // innholdet som en tegnstreng
byte[] byter = ut.toByteArray();                // innholdet som en byte-tabell

System.out.print(innhold + " ");                // skrives som tegnstreng
System.out.println(Arrays.toString(byter));     // skrives som tabell

// Utskrift: ABCDabcd [65, 66, 67, 68, 97, 98, 99, 100]
```

Programkode* A.3.11 *b)

Oppgaver til A.3.11

1. La tabellen *verdier* i *Programkode* A.3.11 *b)* være kilde for en ByteArrayInputStream *inn*. Hent så byter (én og én ved bruk av *read*()) fra *inn* og legg dem fortløpende over i en ByteArrayOutputStream *inn ut* ved hjelp av *write*(). Lag så utskrifter av innholdet slik som i *Programkode* A.3.11 *b)*.

A.3.12 DataOutputStream

DataOutputStream er en direkte subklasse av FilterOutputStream (se Figur A.3.8) og implementerer grensesnittet DataOutput. Klassen er den «omvendte» til DataInputStream Disse to klassene brukes til formatert lesing og skriving. Se Avsnitt A.3.7. De har i tillegg til metodene fra OutputStream, metoder som skriver og leser verdier av alle standardtypene (*byte*, *short*, *int*, *long*, *char*, *float*, *double* og *boolean*).

Klassen har også metoder som skriver ut innholdet av en tegnstreng. Det er:

```
public final void writeBytes(String s) throws IOException
public final void writeChars(String s) throws IOException
public final void writeUTF(String str) throws IOException
```

Hvert tegn i en tegnstreng er en *char* (Java bruker Unicode) på 16 biter eller 2 byter. Metoden *writeBytes*() skriver kun den siste byten i hvert tegn og *writeChars*() skriver hele tegnet (begge bytene). Metoden *writeUTF*() er litt annerledes. Den skriver ut i et modifisert UTF-8 format. I Java er tegn som nevnt representert i Unicode. Grovt sett vil da de tegnene vi normalt bruker (området fra 0 til 255), bli skrevet ut med én byte for tegn fra 1 til 127 og to byter for resten (0 dvs. \u000 og tegn fra 128 til 255. I tillegg skrives det ut informasjon om hvordan dette blir bygget opp. Det er laget slik at det som skrives ut med *writeUTF*() kan leses ved hjelp av *readUTF*() i DataInputStream.

Flg. eksempel viser hvordan vi kan skrive ut formatert vha. metodene i DataOutputStream og lese det vha. metodene i DataInputStream:

```
ByteArrayOutputStream but = new ByteArrayOutputStream();
DataOutputStream ut = new DataOutputStream(but);

ut.writeByte(-1);
ut.writeInt(12345);
ut.writeDouble(3.1415);
ut.writeUTF("ABCÆØÅ");
ut.writeBoolean(false);

System.out.println(but.size());  // Utskrift: 25

ByteArrayInputStream binn = new ByteArrayInputStream(but.toByteArray());
DataInputStream inn = new DataInputStream(binn);

System.out.println(inn.readByte());     // Utskrift: -1
System.out.println(inn.readInt());      // Utskrift: 12345
System.out.println(inn.readDouble());   // Utskrift: 3.1415
System.out.println(inn.readUTF());      // Utskrift: ABCÆØÅ
System.out.println(inn.readBoolean());  // Utskrift: false
```

Programkode A.3.12 a)

Oppgaver til A.3.12

1. Finn ut hvorfor de fem utskriftssetningene tilsammen skriver ut 25 byter. Finn også ut hvordan det i innlesningen dvs. i *readUTF*(), er mulig å avgjøre om det skal leses kun én byte eller to byter?

A.3.13 PrintStream

PrintStream er en subklasse av FilterOutputStream og brukes til lesbar utskrift, dvs. tekst vi kan lese. F.eks. er *out* i klassen System en instans av PrintStream. Den bruker vi jo når vi ønsker å lage utskrift på direkten (til konsollet) fra et frittstående Java-program. Klassen har flg. seks konstruktører:

```
1.  Printstream(File file)
2.  Printstream(OutputStream out)
3.  Printstream(OutputStream out, boolean autoFlush)
4.  Printstream(OutputStream out, boolean autoFlush, String encoding)
5.  Printstream(String fileName)
6.  Printstream(String fileName, String csn)
```

I et programmeringsmiljø er det normalt mulig å bestemme hva som skal være standard tegnsett, f.eks. UTF-8 eller ISO-8859-1. Dermed vil utskrift skje med det tegnsettet. Men i to av konstruktørene kan en selv velge tegnsett for utskrift. Det er nok den 5. konstruktøren som er mest brukt:

```
PrintStream ut = new PrintStream("utskrift.txt");
```

Setningen over vil kaste et unntak hvis filen ikke lar seg opprette. Hvis derimot filen finnes fra før, vil den bli trunkert til å ha null innhold. Ved utskrift til fil er det vanlig å bruke en bufferteknikk. Det gjøres automatisk (ved bruk av en BufferedWriter) i alle konstruktørene til PrintStream. Vi trenger derfor ikke å tenke på det.

Det er fire grupper av utskriftsmetoder. Det er for det første *write*-metodene som arves fra OutputStream. De oppfører seg slik vi er vant til. I prinsippet kunne vi ha klart oss med dem. F.eks. kunne vi få skrevet ut en tegnstreng på denne måten:

```
PrintStream ut = new PrintStream("utskrift.txt");

String s = "Dette er en tegnstreng!";
ut.write(s.getBytes());

ut.close();
```

Programkode A.3.13 a)

Men *write*-metodene brukes normalt lite i PrintStream. Det er *print*-metodene (*print* og *println*) som det er vanlig å bruke. Klassen har isteden fått de to private metodene void *write*(char buf[]) og void *write*(String s). De to typene *print*-metoder er kodet ved hjelp av dem. F.eks. er de to *print*-metodene som skriver ut et heltall (på desimalform), kodet slik (der *newLine*() sender ut et linjeskifte):

```
public void print(int heltall)
{
  write(String.valueOf(heltall));  // fra int til desimale siffer
}

public void println(int heltall)
{
  print(heltall);  // kaller metoden over
  newLine();       // linjeskift
}
```

Programkode A.3.13 b)

Det vi ser rett over gjelder alle *println*-metodene, dvs. at de kun består av et kall på den tilsvarende *print*-metoden pluss et kall på *newLine*(). Obs: Hva metoden *newLine*() sender ut er plattformavhengig. I et Windows-miljø er det to byter, dvs. CR (Carriage Return = 13) og LF (Line Feed = 10). I et annet miljø kan det være kun CR eller kun LF.

Det er *print*-metoder (og *prinln*-metoder) for datatypene *int*, *long*, *float*, *double*, *char*, *boolean*, *Object*, *String* og *char*[]. Det er ingen for datatypene *byte* og *short*, men det trengs ikke siden *int*-versjonen virker for dem. Metodekallet *print*(), dvs. uten parameter, er ulovlig. Men *println*() er ok siden det kun inneholder et kall på *newLine*() og gir dermed et linjeskifte. Det er også mulig å få til et linjeskifte ved å skrive ut tegnet '\n', f.eks. ved *print*('\n'). Da er det tegnet LF (10) som skrives ut.

Siden System.out er en instans av PrintStream, er det enkelt å bruke den til å teste *print*-metodene. I flg. eksempel har vi informasjon om studenter: navn (fornavn), studiepoeng og alder. Dette skal skrives ut med en student per linje:

```
String[] navn = {"Petter", "Kari", "Aleksander", "Elin"};
int[] poeng = {10, 140, 90, 70};
int[] alder = {19, 21, 20, 19};

for (int i = 0; i < navn.length; i++)
{
  System.out.println(navn[i] + " " + poeng[i] + " " + alder[i]);
}

// Petter 10 19
// Kari 140 21
// Aleksander 90 20
// Elin 70 19
```

Programkode* A.3.13 *b)

Her er det *println*(String) som brukes siden argumentet blir konvertert til en tegnstreng. Men dette er ikke bra nok hvis målet er å få utskriften formatert, dvs. informasjonen i pene kolonner. Hva hvis vi f.eks. ønsker flg. utskrift:

```
Petter       10  19
Kari        140  21
Aleksander   90  20
Elin         70  19
```

Dette kan vi få til ved litt arbeid. Hvis dette skal være et generelt program og ikke kun for disse fire studentene, må vi normalt tenke på hvor langt et fornavn kan være. Men her gjør vi det enkelt og sier at et fornavn ikke kan ha mer enn ti tegn (som i Aleksander). Videre skal det være mellomrom på to mellom hver kolonne:

```
for (int i = 0; i < navn.length; i++)
{
  System.out.print(navn[i]);
  for (int j = 0; j < 12 - navn[i].length(); j++) System.out.print(" ");
  if (poeng[i] < 100) System.out.print(" ");
  System.out.println(poeng[i] + "  " + alder[i]);
}
```

Programkode* A.3.13 *d)

Her er det tatt hensyn til at lengden på navn og antall siffer i studiepoeng kan variere. Det skrives ut så mange mellomrom at vi får rette kolonner. Men dette er fortsatt ikke fleksibelt nok. Hvordan blir utskriften hvis f.eks. Petter har 0 studiepoeng og ikke 10?

Det er også en versjon av *print*() (og av *println*()) som har referansetypen Object som parametertype. Hvis vi da bruker en instans av en eller klasse som argument, er det instansens *toString*() som blir brukt.

Den tredje gruppen av utskriftsmetoder er de to *printf*-metodene. Vi ser på den ene av de to. Den har flg. syntaks:

```
public PrintStream printf(String format, Object... args)
```

En nærmere forklaring på argumentlisten finner du i *Vedlegg* B. Her skal vi bruke metoden til å forenkle *Programkode* A.3.13 *d)*:

```
for (int i = 0; i < navn.length; i++)
{
  System.out.printf("%-10s  %3d  %2d\n", navn[i], poeng[i], alder[i]);
}
```

Programkode* A.3.13 *e)

Koden over gir «pen» utskrift også hvis Petter har 0 studiepoeng.

Den fjerde og sist gruppen av utskriftsmetoder er *append*-metodene. Systemet har en utskriftskø. Den tømmes når vi lukker (close) eller eksplisitt ber om det (flush). En *append* legger verdier inn på slutten av køen. Det er tre *append*-metoder:

```
PrintStream append(char c)
PrintStream append(CharSequence csq)
PrintStream append(CharSequence csq, int start, int end)
```

Her inngår en CharSequence som parameter. Både String og StringBuilder implementerer dette grensesnittet. Det betyr spesielt at vi kan la en String være parameter, men ikke heltall og desimaltall. De må først konverteres til en String. Flg. kode gir den samme utskriften som i *Programkode* A.3.13 *b)*:

```
for (int i = 0; i < navn.length; i++)
{
  System.out.append(navn[i]).append(' ').
     append(String.valueOf(poeng[i])).append(' ').
     append(String.valueOf(alder[i])).append('\n');
}
```

Programkode* A.3.13 *f)

Oppgaver til A.3.13

1. Hva skjer med *Programkode* A.3.13 *e)* hvis vi har et navn som har flere enn 10 tegn? Lag kode som gjør at slike navn trunkeres, dsv. at kun de 10 første tas med.

Copyright © Ulf Uttersrud, 2018. All rights reserved.