A.1 BitInputStream

A.1.1 Instansiering

`BitInputStream` har fire konstruktører og to konstruksjonsmetoder (eng: factory methods):

```
public BitInputStream(InputStream in)                    // konstruktør
public BitInputStream(InputStream in, int size)          // konstruktør
public BitInputStream(String fileName)                   // konstruktør
public BitInputStream(String filename, int size)         // konstruktør
public static BitInputStream fromFile(String fileName)   // konstruksjonsmetode
public static BitInputStream fromByteArray(byte[] b)     // konstruksjonsmetode
```

Programkode* A.1.1 *a)

`BitInputStream` bruker et internt buffer for å gjøre fil-lesingen mest mulig effektiv. Når bufferet er tomt blir det fylt ved hjelp av kun én leseoperasjon. Dermed blir antallet leseoperasjoner kraftig redusert. En fil kan ligge på en harddisk, på en CD, på en minnepinne eller kanskje på internett. Operasjoner på slike ytre enheter går alltid vesentlig saktere enn operasjoner internt i datamaskinen. Derfor er det lønnsomt å ha få leseoperasjoner.

Konstruktørene trenger en `InputStream` eller et filnavn som parameter. Det interne bufferet har 4kb (4096 byter) som standardstørrelse, men det kan velges (parameteren *size*).

De to konstruksjonsmetodene er litt annerledes. Den første knytter en `BitInputStream` til en fil med oppgitt filnavn, og bruker standard bufferstørrelse. Den andre bruker en byte-tabell som den bitsekvensen det skal leses fra. Der blir bufferstørrelsen satt lik byte-tabellens lengde så sant den ikke er større enn standard bufferstørrelse.

Hvis vi skal opprette en instans av `BitInputStream` knyttet til en fil med navn `"innfil.txt"`, kan vi gjøre det på en av flg. to måter (OBS: begge måtene kaster en IOException):

```
public static void main(String... args) throws IOException
{
  String innfil = "innfil.txt";
  BitInputStream inn1 = new BitInputStream(innfil);       // konstruktør
  BitInputStream inn2 = BitInputStream.fromFile(innfil);  // konstruksjonsmetode
}
```

Programkode* A.1.1 *b)

Koden over virker hvis `innfil.txt` ligger på gjeldende (eng: current) område. Men vi kan også oppgi hele veien (eng: path):

```
String innfil = "c:/NetBeansAlgDat/AlgDat/innfil.txt";
BitInputStream inn1 = new BitInputStream(innfil);       // konstruktør
BitInputStream inn2 = BitInputStream.fromFile(innfil);  // konstruksjonsmetode
```

Programkode* A.1.1 *c)

Programkode A.1.1 *b)* og *c)* virker for lokale filer. Men vi kan også hente filer fra internett ved hjelp av en instans av klassen `URL` (se også *Delkapittel* A.5). Da kan vi bruke den første konstruktøren i *Programkode* A.1.1 *a)*:

```
BitInputStream inn = new BitInputStream((new URL("http://www.vg.no")).openStream());
```

Programkode A.1.1 d)

Det er mulig å hente en lokal fil ved hjelp av en url. Da må filveien starte med `"file:///"`.

```
String innfil = "file:///c:/NetBeansAlgDat/AlgDat/innfil.txt";
BitInputStream inn = new BitInputStream((new URL(innfil)).openStream());
```

Programkode A.1.1 e)

Når vi senere skal teste metodene i klassen `BitInputStream`, vil det være gunstig på forhånd å kjenne til det nøyaktige bitinnholdet i en fil. Gitt at vi har flg. sekvens eller strøm av biter:

```
01100111010110010110111011100101011000101001110 0
```

Figur A.1.1 a)

Det er tilsammen 48 biter. Vi kunne f.eks. tenke oss at det er de første 48 bitene (dvs. de første 7 bytene) på filen `"innfil.txt"`. Men det er også mulig å lage en byte-tabell som gir oss den samme sekvensen. De første 8 bitene er `01100111`. Dette kan tolkes som tallet 103 (= 64 + 32 + 4 + 2 + 1). De neste 8 bitene er `01011001` og det blir 89 (= 64 + 16 + 8 + 1). Osv. Dermed vil flg. byte-tabell gi oss bitsekvensen i *Figur* A.1.1 *a)*:

```
byte[] b = {103, 89, 110, -27, 98, -100};
```

Vi kan opprette en instans av `BitInputStream` knyttet til byte-tabellen *b* på en av flg. måter:

```
BitInputStream inn = new BitInputStream(new ByteArrayInputStream(b),b.length);
BitInputStream inn = BitInputStream.fromByteArray(b);
```

Programkode A.1.1 f)

Her blir `b.length` bufferstørrelse (i det andre tilfellet implisitt i konstruksjonsmetoden). Det har ingen hensikt å bruke en bufferstørrelse som er større enn størrelsen på byte-tabellen.

I flg. eksempel brukes metoden *readBits*() (vi ser nærmere på den i *Avsnitt* A.1.2):

```
byte[] b = {103,89,110,-27,98,-100};  // bitsekvensen i Figur A.1.1 a)
BitInputStream inn = BitInputStream.fromByteArray(b);  // se Programkode A.1.1 a)
int biter = 0;
while ((biter = inn.readBits(6)) != -1)  // 6 biter om gangen
  System.out.print(biter + " ");         // biter er et heltall
inn.close();  // Utskrift: 25 53 37 46 57 22 10 28
```

Programkode A.1.1 g)

De seks første bitene i *Figur* A.1.1 *a)* er 011001. Deretter kommer 110101. Tolket som heltall blir det, slik som også utskriften viser, lik 25 og 53.

En tegnstreng kan tolkes som en bitsekvens siden hvert tegn har 8 biter. Vi leser (denne gangen ved å bruke en for-løkke) ved hjelp av flg. konstruksjon:

```
String s = "Dette er den første delen av en tekst.";
BitInputStream inn = BitInputStream.fromByteArray(s.getBytes());

for (int biter = inn.readBits(8); biter != -1; biter = inn.readBits(8))
  System.out.print((char)biter);         // gjør om til tegn
inn.close();  // Utskrift: Dette er den første delen av en tekst.
```

Programkode A.1.1 h)

A.1.2 Read

Vi antar nå at vi har en instans av `BitInputStream` ved navn *inn* og at de første bitene i den tilhørende bitsekvensen er som i *Figur* A.1.1 *a)* (f.eks. ved hjelp *Programkode* A.1.1 *f*):

`BitInputStream` har tre lesemetoder:

```
public int readBits(int numberOfBits)  // leser numberOfBits biter
public int read()                      // leser 8 biter
public int readBit()                   // leser 1 bit
```

Før lesingen starter vil bitsekvensen se slik ut:

```
01100111010110010110111011100101011000101001110 0
x
```

Figur* A.1.2 *a)

Symbolet x markerer gjeldende posisjon (eng: current position), og før noe er lest er gjeldende posisjon helt i begynnelsen av sekvensen. Vi leser 17 biter:

```
int n = inn.readBits(17);
// n får verdien  00000000000000001100111010110010
```

Programkode* A.1.2 *a)

De 17 leste bitene vil bli returnert som de 17 bakerste (minst signifikante) bitene i et heltall. Resten (de 15 forreste) vil garantert være 0-biter. I *Programkode* A.1.2 *a)* over vil variablen *n* inneholde de 17 første bitene fra sekvensen og de ligger bakerst (markert med fet type). I tillegg vil gjeldende posisjon ha flyttet seg 17 posisjoner mot høyre slik som *Figur* A.1.2 *b)* under viser:

```
01100111010110010110111011100101011000101001110 0
                  x
```

Figur* A.1.2 *b)

Vi kan gå videre fra *Programkode* A.1.2 *a*):

```
n = inn.readBits(5);
// n får verdien  00000000000000000000000000011011
```

Programkode* A.1.2 *b)

Variablen *n* innholder nå de 5 neste bitene og gjeldende posisjon er flyttet 5 posisjoner mot høyre slik som *Figur* A.1.2 *c)* under viser:

```
01100111010110010110111011100101011000101001110 0
                        x
```

Figur* A.1.2 *c)

Generelt gjelder at metoden `int` *readBits*(`int` *numberOfBits*) leser et oppgitt antall biter (fra og med og gjeldende posisjon) og bitene blir returnert som de bakerste bitene i et heltall. Resten av heltallet (de forreste bitene) består av 0-biter. Internt vil gjeldende posisjon ha flyttet seg like mange posisjoner mot høyre som antallet leste biter. Det er imidlertid en restriksjon. Det er ikke mulig å lese flere enn 31 biter på en gang. Grunnen til at det ikke kan leses 32 biter er at -1 (end of file) returneres når det er færre enn *numberOfBits* biter igjen. I så fall ville det ikke være mulig å skille mellom når -1 (dvs. 32 1-biter) representerer bitene i en innlesing eller når -1 signaliserer at at det ikke er nok biter igjen å lese.

Den neste metoden - `int read()` - leser nøyaktig 8 biter (1 byte) (og gjeldende posisjon flyttes 8 posisjoner mot høyre). Den er tatt med fordi klassen `InputStream` har en slik metode. F.eks. vil nå koden *read*() og koden *readBits*(8) ha samme effekt. Den siste metoden - `int readBit()` - leser én bit og returnerer 0 eller 1 avhengig av om den leste biten var en 0- eller 1-bit. Internt flyttes gjeldende posisjon en mot høyre. Spesielt vil *readBit*() og *readBits*(1) ha samme effekt, men den første er litt raskere.

Felles for alle de tre *read*-metodene er at -1 returneres hvis det er færre enn det ønskede antallet biter igjen i sekvensen. Dette behøver ikke å bety at vi har kommet til slutten på sekvensen (end of file). Hvis det er f.eks. 5 biter igjen i sekvensen og vi ber om å få lese 10, vil vi få -1 som returverdi.

I flg. eksempel opprettes (som i *Programkode* A.1.1 *d*) en `BitInputStream` knyttet til en byte-tabell med 6 byter = 48 biter. Vi leser så 10 biter om gangen. Da vil det være 8 biter igjen. Men generelt kan vi bruke metoden *availableBits*() til å finne ut det:

```
public static void main(String... args) throws IOException
{
  byte[] b = {103,89,110,-27,98,-100};  // bitsekvensen i Figur A.1.1 a)
  BitInputStream inn = BitInputStream.fromByteArray(b);
  int k = 10;

  for (int biter = inn.readBits(k); biter != -1; biter = inn.readBits(k))
  {
    System.out.print(biter + " ");        // skriver ut bitene som et heltall
  }

  int r = inn.availableBits();            // antall biter som er igjen
  if (r > 0)
    System.out.println(inn.readBits(r));  // leser resten

  inn.close();  // Utskrift: 413 406 953 354 156
}
```

Programkode* A.1.2 *c)

Oppgaver til A.1.2

1. Bruk andre verdier enn *k* = 10 i *Programkode* A.1.2 *c)* over. Da vil enhver verdi på *k* fra 1 til 31 kunne brukes.
2. Det er ikke mulig å lese inn mer enn 31 biter om gangen. Men en kan omgå dette hvis en f.eks. leser 16 biter og så skjøter sammen dette til en int med de 16 bitene i første innlesing forrest og de neste 16 bitene bakerst. Ta utgangspunkt i byte-tabellen b i *Programkode* A.1.2 *c)* over og gjør dette for de første 32 bitene. Gjenta dette f.eks. med 31 biter i første innlesing og 1 bit i andre. Eller andre kombinasjoner der summen blir 32. Prøv også med 32 som *numberOfBits* i metoden *read*(). Hva skjer?
3. Bytt ut alle verdiene i byte-tabellen i *Programkode* A.1.2 *c)* over med -1. Gjør da på samme måte som i Oppgave 1.

A.1.3 Peek og unread

Det vil være situasjoner der det er ønskelig å se på biter i bitsekvensen uten at de tas ut, dvs. at gjeldende posisjon ikke endres. Det er tre slike metoder:

```
public int peek()                       // ser på de 8 neste bitene
public int peekBit()                    // ser på neste bit
public int peekBits(int numberOfBits)   // ser på de numberOfBits neste bitene
```

Metodene over returnerer det samme som de tilsvarende *read*-metodene, men som nevnt flyttes ikke gjeldende posisjon. Det betyr at vi får samme returverdi hvis et *peek*-kall gjentas. Metodene returnerer -1 hvis det er færre biter igjen i bitsekvensen enn det antallet som vi ønsker å se på.

Etter en *read* kan noen eller alle bitene som ble lest, legges tilbake (eng: push back). Det som egentlig skjer er at gjeldende posisjon flyttes tilbake (mot venstre). Obs: Det kan ikke legges tilbake biter etter en *peek* eller en *skip*. Vi har fire *unread*-metoder:

```
public void unreadBit()                  // legger tilbake 1 bit
public void unreadBits()                 // legger tilbake så mange som mulig
public void unreadBits(int numberOfBits) // legger tilbake numberOfBits biter
public void unread()                     // legger tilbake 8 biter
```

Et kall på en *unread* krever som nevnt at det har vært en *read* rett foran. Etter en *read* kan en eller flere av de bitene som ble lest, legges tilbake, men ikke flere enn de som ble lest. Av implementasjonstekniske årsaker kan det være situasjoner der maksimum er 25 biter. Men generelt vil metoden *unreadSize*() fortelle hvor mange biter som kan legges tilbake. En *unread* kan kalles flere ganger på rad hvis det sammenlagt ikke legges tilbake flere biter enn de som ble lest ved siste *read*. Det kastes et unntak hvis dette brytes.

Legg merke til at en kombinasjon av en *read* og en *unread* gir samme effekt som en *peek*. Men hvis formålet kun er å se på et bestemt antall biter, så er *peek* mest effektivt.

I flg. eksempel tar vi på nytt utgangspunkt i bitsekvensen *figur* A.1.1 *a)*.

```
public static void main(String... args) throws IOException
{
  //0110011101011001011011101110010101100010100111100
  byte[] b = {103,89,110,-27,98,-100};  // gir bitsekvensen over
  BitInputStream inn = BitInputStream.fromByteArray(b);

  int k = inn.readBits(13);  // leser 13 biter, dvs. 0110011101011
  inn.unreadBits(5);         // legger de 5 siste tilbake, dvs. 01011
  k = inn.peek();            // ser på de 8 første, dvs. 01011001
  inn.unreadBit();           // feil! - unread ulovlig etter en peek

  inn.close();
}
```

Programkode* A.1.3 *a)

Oppgaver til A.1.3

1. Sjekk at *Programkode* A.1.3 *a)* over virker, dvs. kaster et unntak. Ta så vekk setningen som forårsaker unntaket.
2. Etter en *read* kan det garantert legges tilbake (eng: push back) så mange biter som det maksimale av 25 og det antallet som ble lest. Men mange ganger mer. Ta utgangspunkt i de to første linjene i *Programkode* A.1.3 *a)*. Lag så kode som først leser inn 2 og så 31 biter. Sjekk så hva metoden *unreadSize*() returnerer. Les så isteden inn kun én bit første gang. Hva da?

A.1.4 Skip og insert

Det er mulig å hoppe over (eller skippe, eng: skip) et antall biter. Det betyr at det eneste som skjer er gjeldende posisjon i bitsekvensen flyttes (mot høyre). Vi har fire *skip*-metoder:

```
public int skipBit()                        // hopper over 1 bit
public int skip()                           // hopper over 8 biter
public int skipBits(int numberOfBits)       // hopper over numberOfBits biter
public long skipBit(int long)               // UnsupportedOperationException
```

De tre første *skip*-metodene hopper over (skipper) så mange biter som nevnt over. Antallet biter som det ble hoppet over (eller skippet) returneres. Det er ikke mulig å hoppe lenger i bitsekvensen enn til slutten. Hvis det er færre biter igjen enn det ønskede antallet, returneres det aktuelle antallet. Det betyr spesielt at hvis vi allerede har nådd til slutten av bitsekvensen, vil returverdien bli 0 uansett hvilken *skip* som kalles. Hvis en forsøker å bruke den siste av de fire *skip*-metodene, vil det bli kastet en `UnsupportedOperationException`. Den arves fra basisklassen `InputStream` hvor den er kodet. Men den koden gir ikke mening her. Obs: Etter en *skip* er det ikke tillatt med en *unread*!

Eksempel:

```
public static void main(String... args) throws IOException
{
  //01100111010110010110111011100101011000101001 0011100
  byte[] b = {103,89,110,-27,98,-100};  // gir bitsekvensen over
  BitInputStream inn = BitInputStream.fromByteArray(b);

  inn.skipBits(24);                     // hopper over 24 biter (halvparten)
  inn.readBits(12);                     // nå er det igjen 12 biter
  System.out.println(inn.skipBits(24)); // Utskrift: 12

  inn.close();
}
```

Programkode A.1.4 *a)*

Det kan settes inn biter i bitsekvensen, dvs. biter foran de som allerede ligger der:

```
public void insert1Bit()                            // setter inn en 1-bit
public void insert0Bit()                            // setter inn en 0-bit
public void insertBit(int bit)                      // setter inn en bit
public void insert()                                // setter inn 8 biter
public void insertBits(int value, int numberOfBits) // inn numberOfBits biter
```

Metodene `public void insertBit(int bit)` setter inn den siste biten fra argumentet *bit* og `public void insertBits(int value, int numberOfBits)` de *numberOfBits* siste bitene fra argumentet *value*.

Det er restriksjoner på hvor mange nye biter det kan settes inn. Rett etter en *read* kan det normalt settes inn minst så mange biter som ble lest. Men i noen situasjoner kan det maksimale antallet være 25 biter. Metoden *insertSize*() forteller til enhver tid hvor mange biter som kan settes inn. Obs: Etter en *insert* gir *unreadSize*() en tilsvarende mindre verdi.

Oppgaver til A.1.4

1. Ta utgangspunkt i byte-tabell og BitInputStream i *Programkode* A.1.4 *a)*. Legg inn en 1-bit først (*insert1bit*) og les så 9 biter. Hva blir resultatet?

A.1.5 Øvrige metoder

Hvis en ønsker å se på bitinnholdet i et heltall, kan en bruke metoden *toBinaryString*() fra klassen Integer. Men den gir de signifikante bitene, dvs. alle bitene hvis det er et negativt tall og alle bitene fra og med første 1-bit ellers:

```
String biter1 = Integer.toBinaryString(12345);    // et positivt tall
String biter2 = Integer.toBinaryString(-12345);   // et negativt tall
System.out.println(biter1 + " " + biter2);
// Utskrift: 11000000111001 11111111111111111100111111000111
```

Programkode **A.1.5** *b)*

Men denne metoden fungerer ikke hvis vi ønsker å få ut et bestemt antall biter eller eventuelt et intervall av biter. Klassen BitInputStream har to metoder for disse formålene. Den første har *numberOfBits* som argument og gir så mange av de bakerste (siste) bitene (og kaster et unntak hvis *numberOfBits* er negativ eller er større enn 32):

```
String biter1 = Integer.toBinaryString(12345);
String biter2 = BitInputStream.toBinaryString(12345, 14);
System.out.println(biter1 + " " + biter2);
// Utskrift: 11000000111001 11000000111001
```

Programkode **A.1.5** *b)*

Metoden kan f.eks. brukes til å sjekke at en read-metode leser og returnerer det som forventes. I *Programkode* A.1.1 *g)* leses det 6 biter om gangen og resultatet skrives ut som heltall. Nå kan vi isteden skrive ut bitene (som en string):

```
byte[] b = {103,89,110,-27,98,-100};  // bitsekvensen i Figur A.1.1 a)
BitInputStream inn = BitInputStream.fromByteArray(b);  // se Programkode A.1.1 a)

int biter = 0;
while ((biter = inn.readBits(6)) != -1)  // 6 biter om gangen
  System.out.print(BitInputStream.toBinaryString(biter, 6) + " ");
inn.close();

// Utskrift: 011001 110101 100101 101110 111001 010110 001010 011100
```

Programkode **A.1.5** *c)*

Den andre metoden har *from* og *to* som argumenter. Da tenker vi oss at bitene har indekser fra 0 til 31 og at de er orientert fra venstre mot høyre. Hvis vi f.eks. ønsker alle de 32 bitene, må *from* = 0 og *to* = 32. Det kastes et unntak hvis *from* er negativ, *from* større enn *to* og *to* større enn 32.

```
int k = 123;
System.out.println(Integer.toBinaryString(k));             // 1111011
System.out.println(BitInputStream.toBinaryString(k, 10));  // 0001111011
```

Programkode **A.1.5** *d)*

Metoden **public static** String ***toBinaryString***(**int** value, **int** from, **int** to) gir bitene i det halvåpne intervallet [from:to> der de 32 bitene er orientert fra venstre mot høyre. Vi kan også bruke den til å finne de 10 siste bitene i *k* = 123:

```
int k = 123;
System.out.println(BitInputStream.toBinaryString(k, 22, 32));  // 0001111011
```

Programkode **A.1.5** *e)*

A.1.6 Metodeoversikt

```
public BitInputStream(InputStream in)                   // konstruktør
public BitInputStream(InputStream in, int size)         // konstruktør
public BitInputStream(String fileName)                  // konstruktør
public BitInputStream(String filename, int size)        // konstruktør
public static BitInputStream fromFile(String fileName)  // konstruksjonsmetode
public static BitInputStream fromByteArray(byte[] b)    // konstruksjonsmetode

public int readBits(int numberOfBits)      // leser numberOfBits biter
public int read()                          // leser 8 biter
public int readBit()                       // leser 1 bit
public int availableBits();                // antall biter igjen

public int peek()                          // ser på de 8 neste bitene
public int peekBit()                       // ser på neste bit
public int peekBits(int numberOfBits)      // ser på de numberOfBits neste bitene

public void unreadBit()                    // legger tilbake 1 bit
public void unreadBits()                   // legger tilbake så mange som mulig
public void unreadBits(int numberOfBits)   // legger tilbake numberOfBits biter
public void unread()                       // legger tilbake 8 biter
public int unreadSize();                   // maks antall som kan legges tilbake

public int skipBit()                       // hopper over 1 bit
public int skip()                          // hopper over 8 biter
public int skipBits(int numberOfBits)      // hopper over numberOfBits biter
public long skipBit(int long)              // UnsupportedOperationException

public void insert1Bit()                   // setter inn en 1-bit
public void insert0Bit()                   // setter inn en 0-bit
public void insertBit(int bit)             // setter inn en bit
public void insert()                       // setter inn 8 biter
public void insertBits(int value, int numberOfBits)  // numberOfBits biter
public int insertSize();                   // maks antall som kan settes inn

// henter ut et gitt antall biter fra et heltall av typen int
public static String toBinaryString(int value, int numberOfBits);
public static String toBinaryString(int value, int from, int to);

public void close();                       // lukker filen
```